เทคนิคการวิเคราะห์เพื่อเพิ่มความสามารถในการทดสอบคลาสคอมโพเนนท์

นางสาวสุภาภรณ์ กานต์สมเกียรติ

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรดุษฎีบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2549

ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

AN ANALYSIS TECHNIQUE TO INCREASE TESTABILITY OF CLASS-COMPONENT

Miss Supaporn Kansomkeat

A Dissertation Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy Program in Computer Engineering
Department of Computer Engineering
Faculty of Engineering
Chulalongkorn University
Academic year 2006
Copyright of Chulalongkorn University

Thesis Title	AN ANALYSIS TECHNIQUE TO INCREASE TESTABILITY OF CLASS-COMPONENT
By	Miss Supaporn Kansomkeat
Field of study	Computer Engineering
Thesis Advisor	Associate Professor Wanchai Rivepiboon, Ph.D.

Accepted by the Faculty of Engineering, Chulalongkorn University in Partial Fulfillment of the Requirements for the Doctoral Degree

……………………………………………….. Dean of the Faculty of Engineering

(Professor Direk Lavansiri, Ph.D.)

THESIS COMMITTEE

……………………………………………….. Chairman

(Associate Professor Somchai Prasitjutrakul, Ph.D.)

……………………………………………….. Thesis Advisor

(Associate Professor Wanchai Rivepiboon, Ph.D.)

……………………………………………….. Member

(Assistant Professor Somnuk Keretho, Ph.D.)

……………………………………………….. Member

(Assistant Professor Taratip Suwannasart, Ph.D.)

……………………………………………….. Member

(Assistant Professor Twittie Senivongse, Ph.D.)

สุภาภรณ์ กานต์สมเกียรติ : เทคนิคการวิเคราะห์เพื่อเพิ่มความสามารถในการทดสอบคลาสคอมโพเนนท์. (AN ANALYSIS TECHNIQUE TO INCREASE TESTABILITY OF CLASS-COMPONENT) อ. ที่ปรึกษา: รศ. ดร.วันชัย ริ้วไพบูลย์, 67 หน้า.

การทดสอบซอฟต์แวร์จะกระทำได้ง่ายขึ้นเมื่อซอฟต์แวร์มีค่าความสามารถในการทดสอบสูง เป็นที่ทราบกันว่า การเพิ่มค่าความสามารถในการทดสอบจะทำให้ความผิดพร่องถูกค้นหาได้อย่างมีประสิทธิภาพมากขึ้น ซอฟต์แวร์เชิงคอมโพเนนท์นั้น โดยมากถูกพัฒนาขึ้นโดยนำคอมโพเนนท์ที่พัฒนาแล้วมาประกอบเข้าด้วยกัน การนำคอมโพเนนท์กลับมาใช้ซ้ำมีความจำเป็นต้องทำการทดสอบอีกครั้งเพื่อให้เกิดความมั่นใจว่าคอมโพเนนท์สามารถทำงานร่วมกันในระบบใหม่ได้อย่างราบรื่น การเพิ่มและการวัดค่าความสามารถในการทดสอบคอมโพเนนท์ที่นำกลับมาใช้จะทำให้การทดสอบคอมโพเนนท์และการค้นหาความผิดพร่องกระทำได้ดีขึ้น ข้อมูลที่ได้จากการวิเคราะห์รหัสต้นฉบับเป็นส่วนสำคัญของการเพิ่มและการวัดค่าความสามารถในการทดสอบ อย่างไรก็ดีคุณสมบัติของคอมโพเนนท์โดยทั่วไปคือไม่สามารถหารหัสต้นฉบับได้ ดังนั้นจึงต้องทำการวิเคราะห์คอมโพเนนท์ในระดับไบต์เพื่อให้ได้มาซึ่งข้อมูลต่างๆ ที่จำเป็นสำหรับการเพิ่มและวัดค่าความสามารถในการทดสอบ

ข้อมูลที่ได้จากการวิเคราะห์คอมโพเนนท์ในระดับไบต์ สามารถใช้เพิ่มและวัดค่าความสามารถในการทดสอบโดยไม่จำเป็นต้องใช้รหัสต้นฉบับ ขั้นตอนการวิเคราะห์ถูกกระทำเพื่อเก็บรวมรวมข้อมูลของลำดับการทำงานและลำดับการไหลของข้อมูลภายในคอมโพเนนท์ ข้อมูลลำดับการทำงานและการไหลเหล่านี้ถูกนำไปใช้เพื่อหาข้อมูลเกี่ยวกับการกำหนด และการใช้ตัวแปรต่างๆ ของแต่ละเมท็อดในคอมโพเน้นท์ ข้อมูลการกำหนดและการใช้ตัวแปรที่ได้ จะถูกนำไปใช้เพื่อเพิ่มความสามารถในการทดสอบ และวัดค่าความสามารถในการทดสอบซึ่งนำไปสู่การประเมินความยุ่งยากของขั้นตอนการทดสอบในที่สุด

งานวิจัยนี้ได้พัฒนาและทดลองวิธีการเพิ่มและวัดค่าความสามารถในการทดสอบของคอมโพเน้นท์ ผลการทดลองแสดงให้เห็นว่าวิธีการเพิ่มความสามารถในการทดสอบที่ได้นำเสนอนี้ ทำให้ความสามารถในการค้นพบความผิดพร่องเพิ่มขึ้น นอกจากนั้นยังพบว่า การวัดค่าความสามารถในการทดสอบที่ได้นำเสนอนี้มีประสิทธิภาพในการบ่งชี้ความสามารถในการทดสอบของคอมโพเน้นท์

ภาควิชา.......วิศวกรรมคอมพิวเตอร์.........ลายมือชื่อนิสิต...
สาขาวิชา......วิศวกรรมคอมพิวเตอร์.........ลายมือชื่ออาจารย์ที่ปรึกษา.......................................
ปีการศึกษา...2549...............

4471826221 : MAJOR COMPUTER ENGINEERING
KEY WORD: SOFTWARE TESTING / SOFTWARE TESTABILITY / TESTABILITY ANALYSIS

SUPAPORN KANSOMKEAT: AN ANALYSIS TECHNIQUE TO INCREASE TESTABILITY OF CLASS-COMPONENT. THESIS ADVISOR: ASSOC. PROF. WANCHAI RIVEPIBOON, Ph.D., 67 pp.

Testing software is made easier when testability is high. In general, increasing testability allows faults to be detected more efficiently. Component-based software is often constructed from third party software components. When this is done, the reused components must be retested in the new environment to ensure that they integrate correctly into the new context. Therefore, it is helpful to increase a component's testability before it is reused. Also, it is helpful to measure a component's testability to indicate the difficulty of revealing faults. Some information about components is needed to increase and measure component's testability, such as information gotten through program analysis. A crucial property of reused software components is that the source is not available, making program analysis significantly more difficult. This thesis addresses this problem by performing program analysis at bytecode level.

This bytecode analysis technique increases and measures component testability without requiring access to the source code. A component's bytecode is analyzed to gather control and data flow information, which is then used to obtain definition and use information of method and class variable of component. Then, the definition and use information is used to increase component testability during component integration, and measure component testability for estimating the difficulty of testing. We have implemented the technique and applied it on some sampled components. Experimental results reveal that fault detection ability can be increased by using our increasing testability process. Also, the proposed testability measurement is appropriate in indicating component testability.

Department…….Computer Engineering…….Student's signature..................................
Field of study…..Computer Engineering…….Advisor's signature..................................
Academic year…2006……………

ACKNOWLEDGEMENTS

First, I am especially deeply grateful to my thesis advisor, Associate Professor Dr. Wanchai Rivepiboon, for his continuous support in the Ph.D program. Without his encouragement and constant guidance, I could not have finished this dissertation.

Besides my advisors, I would like to thank Professor Dr. Jeff Offutt of George Mason University for hosting me during parts of this research. Moreover, he also gave me valuable suggestions and comments. More importantly, he taught me how to write academic papers.

I would like to thank the rest of my thesis committee: Associate Professor Dr. Somchai Prasitjutrakul, Assistant Professor Dr. Somnuk Keretho, Assistant Professor Dr. Taratip Suwannasart and Assistant Professor Dr. Twittie Senivongse, who gave insightful comments and reviewed my work.

I would like to thank Thailand's Commission on Higher Education, Ministry of Education for funding. This fund supported for studying in Ph.D. program and gave me a chance to have research collaboration in oversea.

Let me also say 'thank you' to all members of the Software Engineering Laboratory for their helps and discussions on my thesis. Thanks also go to Dr. Natenapa Sriharee and Mr. Tawa Kampachua for being my friend thru all my Ph.D. life.

My special thanks go to my family for their support.

CONTENTS

Page

ABSTRACT (THAI)....iv

ABSTRACT (ENGLISH)....v

ACKNOWLEDGEMENTS....vi

CONTENTS....vii

LIST OF TABLES....ix

LIST OF FIGURES....x

CHAPTER

I INTRODUCTION....1

1.1 Motivation....1

1.2 Objective....3

1.3 Scope....3

1.4 Contribution....4

1.5 Research Methodology....4

1.6 Organization of the thesis....5

II BACKGROUD AND RELATED WORKS....6

2.1 Background....6

2.1.1 Java Bytecode Instructions....6

2.1.1.1 Java Class File Format....6

2.1.1.2 Bytecode Instruction Set....8

2.1.2 Software Testing....10

2.1.3 Testability....13

2.1.4 Mutation Testing....14

2.2 Related Works....15

2.2.1 Testing Object-oriented Classes....15

2.2.2 Testability Analysis....16

III CLASS-COMPONENT TESTABILITY....18

3.1 Bytecode-based Class-Component Analysis....18

3.1.1 Determining Basic Blocks....20

3.1.2 Constructing the CDFGs....22

Page

3.1.3 Applying the CDFGs……………………………………………………………......... 28

3.1.4 Definition-Use of Method (DUM)……………………………………………..29

3.2 Increasing Class-Component Testability…………………………………………. 31

3.2.1 Increasing Controllability…………………………………………………….. 31

3.2.2 Increasing Observability……………………………………………………... 37

3.3 Measuring Class-Component Testability…………………………………………. 41

3.3.1 Execution Analysis…………………………………………………………… 41

3.3.2 Propagation Analysis………………………………………………………… 43

IV CASE STUDY……………………………………………………………………………..47

4.1 Increasing Testability Experiment…………………………………………….........47

4.1.1 Experimental Setting and Results…………………………………………… 49

4.2 Measuring Testability Experiment…………………………………………….........54

4.2.1 Experimental Setting and Results…………………………………………… 55

V SUMMARY AND FUTURE WORKS…………………………………………………... 59

5.1 Summary.……………………………………………………………………………...59

5.2 Future Works………………………………………………………………………… 61

REFERENCES…………………………………………………………………………………... 62

APPENDIX…..…………………………………………………………………………….........65

APPENDIX A. Publications……………………………………………………….........66

BIOGRAPHY……………………………………………………………………………………...67

LIST OF FIGURES

Figure Page

2.1 A Simple Java Class and the Bytecode Instructions for Method CalCoeff…………. 7
2.2 Class File Format……………………………………………………………………….. 8
3.1 The Class-Component Testability Process……………………………………………18
3.2 Partitioning Method calCoeff in Figure 2.1 into Basic Blocks……………………..... 21
3.3 The Flows between Basic Blocks of Method calcoeff……………………………….. 23
3.4 The Vending Machine Class…………………………………………………………. 25
3.5 The Bytecode Instructions for Vending Machine…………………………………….. 26
3.6 The CDFGs of Each Method of Vending Machine…………………………………… 27
3.7 Class-Component Analysis Process……………………………………………………29
3.8 The Process of Collection the Definition and Use Information……………………… 30
3.9 The DUCoTs of Vending Machine's Variables………………………………………...32
3.10 Test Requirement Generation Process…………………………………………………34
3.11 Test Case Generation Process………………………………………………………… 36
3.12 Observability Probes Process………………………………………………………….. 39
3.13 The Process of Increasing Class-Componenet Testability………………………….. 40
3.14 Execution Analysis Process………………………………………………………….. 42
3.15 Propagation Analysis Process………………………………………………………….. 45
3.16 The Detail of Execution in Propagation Analysis Process……………………………45
4.1 A Family of Data Flow Testing Criteria…………………………………………………48
4.2 The Test Generation Process for ACU and ACU-O………………………………….. 51
4.3 The *dd* Data Flow Anomaly of Variable *end* of ArrList Calss………………………...54

LIST OF TABLES

Table Page

3.1 Bytecode Instructions Used in the CDFG Construction Process……………………19

3.2 DefUse Information……………......………………………………………………….. 22

3.3 The DUMs of Vending Machine………………………………………………………30

3.4 The All-coupling-uses of Vending Machine's Variables……………………………...33

3.5 Values to Mutate Each Data Type…………………………………………………… 45

4.1 The Description of Each Class Used in the Experiment…………………………….. 51

4.2 The Number of Test Cases of ACU and AllDU tests…………………………………51

4.3 The Number of Mutants for Each Class……………………………………………….. 52

4.4 Case Study Results on All-coupling-uses and All-coupling-uses with Observability……………………………………………………………………… 53

4.5 Case Study Results on All-du-paths and All-coupling-uses with Observability…...53

4.6 The Number of Definition-use Pairs and The Number of Def-clear Path Executions…………………………………………………………………….............. 54

4.7 The Considered Locations of Each Class……………………………………......... 56

4.8 The Number of Test Cases Executing Each Location of Vending Machine………. 57

4.9 The Number of Executions that the execution output of mutant differs from the original output of Vending Machine……………………………………………….. 57

4.10 The Testability Measure of Vending Machine class………………………………..... 58

4.11 The Testability Measure of Each Class………………………………………….......58

CHAPTER I
INTRODUCTION

The purpose of this chapter is to give an overview of the topic of this thesis. We discuss motivation, objective, scope and contribution of this thesis.

1.1 Motivation

Program analysis is a way to inspect programs to gather some properties such as control and data flow information. An early use was to support code optimization in compilers [1]. Program analysis has also been widely used for software engineering problems such as program understanding, testing, and maintenance. Program analysis is used in testing to precisely compute what parts need to be executed [2], to determine which test cases must be rerun to test the program after modifying [3], and to generate more effective tests [4].

Software testing is used to verify software quality and reliability, but it can be an expensive and labor-intensive task [5]. Software testing attempts to reveal software faults by executing the program on inputs and comparing the outputs of the execution with expected outputs. Many research papers have focused on methods to reduce the test effort [6, 7, 8]. An aspect of software that influences the test effort and success is known as *testability*. Testability is defined in terms of *controllability* and *observability*. Controllability is the ease of controlling inputs of tests. Observability is the ease of observing the outputs.

Object-oriented software is increasingly used, partly because it emphasizes portability and reusability. Java classes are compiled into portable binary class files which contain statements called *bytecode*. The class-components are included in Java libraries without source code, thus the source is not always available.

An important goal of reusable components is that the “re-users” should not need to understand how the components work, and should not need or want access to the source. Furthermore, it is usually assumed that the initial developers tested the

component. However, this initial testing was either carried on the component in isolation (unit testing), in its original context, or both.

The goal of this thesis is to test a reused component with regards to how it integrates into a new context. That is, this is a form of integration testing that asks whether the component behaves appropriately in this new context. When a component is reused, a key issue is how to test the component in its new context. Weyuker [9] suggests that a component should be tested many times, individually and also each time it is integrated into a new system. Voas and Miller [10] explained that testability enhances testing and claimed that increasing testability of components is crucial to improving the testability of component-based software. Wang et al. [11] increase component testability by using the built-in test (BIT) approach, that is, putting complete test cases inside the components. The tests are constantly presented and reused with the component. The disadvantage of BIT is growth of programming overhead and component complexity. Naturally, component developers do not always provide BIT and also testing information to component users. Instead of increasing testability by BIT, this research tests the component when it is integrated into a new context. Because of the lack of source code, program analysis techniques cannot be applied to the source. To address this problem, we apply program analysis at the bytecode level.

Component testability analysis and measurement can be used to estimate the difficulty of testing components, aiding planning and execution of testing. Also, testability measure can be used to determine whether the component should be modified to increase its testability before reuse. And it can be used to check the testability improvement of a component after increasing its testability.

This thesis presents an analysis technique to analyze a java class at the bytecode level that is used to directly increases and measures component testability without requiring access to the source. First, Java bytecode (.class file) is analyzed to extract the essential information of control flows and data flows. Then, this flow information is used to collect definition and use information of component's method and

class variables. Finally, the collected information is used to increase and measure component testability. With the above strategy just presented, we considered a java class a component called *class-component*.

The increased testability supports class-component testing by supporting the generation of tests to exercise a class-component in various ways (increasing controllability), thus faults can more easily be revealed. The increased testability also helps to monitor the results of testing (increasing observability), thus class-component failures can more easily be detected. The testability measurement helps to assess a component's testability by measuring the fault revealing ability during testing.

1.2 Objective

The objectives of this thesis are:

- To propose an approach to analyze class-component information and implement the corresponding tool, based on data flow analysis from Java bytecode,
- To design a method to increase testability of class-component and implement the respective tool to support testing by controlling inputs and observing outputs by using the class-component information, and
- To design a method to measure testability of class-component and implement the respective tool by using the class-component information.

1.3 Scope

- In this work, a component is a java class called *class-component*.
- This work proposes an approach (1) to analyze a class-component at bytecode level, (2) to increase testability of class-component and (3) to measure testability of class-component, and implements the corresponding tool.
- This work focuses on intra-class method calls, and does not look for problems that exist in inheritance and polymorphism of multiple classes.
- The efficiency of tests is considered by faults revealing.

- To measure the fault revealing ability, the fault statements are limited to faults in definition and use statements.
- The efficiency of our tests is compared with the all-du-paths criteria by the fault revealing ability based on mutation testing.

1.4 Contribution

The outcome of this thesis will be (1) an analysis mechanism and the corresponding tool for gathering class-component information at bytecode level, and (2) a method and the respective tool for increasing and measuring testability of class-component by using the gathered information.

1.5 Research Methodology

Our research methodologies are presented as follows:

- Review and study the research papers related to component testing, component testability and testability measurement.
- Design an approach to analyze class-component information at bytecode level and implement the corresponding tool.
- Design a method to increase testability of class-component and implement the respective tool.
- Design a method to measure testability of class-component and implement the respective tool.
- Compare the efficiency of our tests with the tests from all-du-paths criteria.
- Evaluate our measurement
- Analyze the results and make conclusions

1.6 Organization of the Thesis

The thesis contents are organized as follows. In Chapter II, we review the background and related works.

In Chapter III, we explain an analysis process to collect information for bytecode instructions. Then, we present a method that uses the colleted information to increase and measure testability of class-component.

Chapter IV presents the case study.

Finally, in Chapter V, we conclude our research work and present some directions for the future work.

CHAPTER II
BACKGROUND AND RELATED WORKS

2.1 Background

This thesis presents an analysis technique to increase and measure class-component testability. The analysis is carried out at the bytecode level. The bytecode instructions are parsed to collect information based on data flow analysis. This information provides ways to generate test inputs and observe the outputs of testing. Coupling-based criteria are used to guide test selection. The tests are evaluated by the fault detection ability based on mutation testing. Thus, this section provides brief overviews of these topics.

2.1.1 Java Bytecode Instructions

Java programs are written and compiled into portable binary class files. Each class is represented by a single file that contains class related data and bytecode instructions. This file is dynamically loaded into an interpreter (Java Virtual Machine, JVM) and executed. An example of a Java program and its corresponding bytecode instructions of method *calCoeff* are shown in Figure 2.1.

2.1.1.1 Java Class File Format

The class file contains simplified seven sections [12] as shown in Figure 2.2.

- Header: Contains the Magic Number OxCAFEBABE and the Version Number.
- Constantpool: Contains all used constants. The first entry is the own class followed by the superclasses, then the constants, and at the end is the name of the source file. The constantpool holds the following types of constants: strings, integers, floats, longs, doubles and references to methods, fields and classes. References are entries in the constantpool which points onto another entry in the constantpool, these entries can be different for each instance.
- Access rights: The access rights of the class.

- Implemented Interfaces: All implemented Interfaces.
- Fields: A list of fields, normally an index into the constantpool.
- Methods: A list of methods, containing the instructions of each method.
- Class Attributes: Contains the source file name and other user definable attributes which are ignored by the JVM.

```
1: public class SetCoeff {
2:
3:   int co,rate;
4:
5:   void calCoeff() {
6:     try {
7:       if (rate == 0 )
8:         throw new  ArithmeticException();
9:       else   if ( rate < 20 )
10:              co = 5;
11:      else
12:              co = 15;
13:     } //try
14:     catch ( ArithmeticException e ) {
15:         System.out.println("Exception");
16:     } //catch
17:   } // calCoeff
18: } // class
```

```
Method void calCoeff()

0: aload_0      //Load var#0 (rate) onto stack
1: getfield      Coeff.rate I (2)
4: ifne   #15   //branch to address 15
7: new  <java.lang.ArithmeticException> (3)
10: dup
11: invokespecial
        java.lang.ArithmeticException.<init> ()V (4)
14:   athrow      //Throwing exception
15:   aload_0
16:   getfield    Coeff.rate I (2)
19:   bipush      20
21:   if_icmpge   #32    //branch to address 32
24:   aload_0
25:   iconst_5
26:   putfield    Coeff.co I (5)
29:   goto        #38   //branch to address 38
32:   aload_0
33:   bipush      15
35:   putfield    Coeff.co I (5)
38:   goto        #50   //branch to address 50
41:   astore_1          //Exception handle block
42:   getstatic
         java.lang.System.out Ljava/io/PrintStream; (6)
45:   ldc         "Exception" (7)
47:   invokevirtual   java.io.PrintStream.println
            (Ljava/lang/String;)V (8)
50:   return
```

Figure 2.1: A Simple Java Class and the Bytecode Instructions for Method CalCoeff

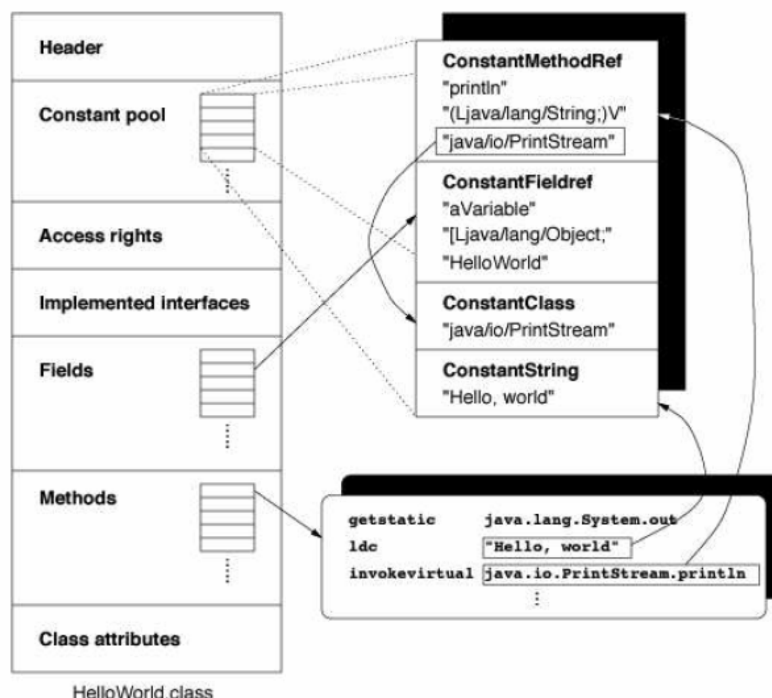

Figure 2.2: Class File Format [12]

2.1.1.2 Bytecode Instruction Set

The Java bytecode instruction set can be roughly grouped as follows:

- *Stack operations*: Constants can be pushed onto the operand stack either by loading them from the constant pool with the *ldc* (load constant) instruction or with special "short-cut" instructions where the operand is encoded into the instructions. For example, *iconst_5* is to push int constant 5 or *bipush* is to push byte value.
- *Arithmetic operations*: The arithmetic instructions compute a result that is typically a function of two values on the operand stack, pushing the result back on the operand stack. The instruction set of the Java Virtual Machine distinguishes its operand types using different instructions to operate on values of specific type. For example, arithmetic operations starting with *i* denote an integer operation such as *iadd* instruction that adds two integers and pushes the result back on the stack.

- *Control flow*: There are unconditional branch instructions like *goto* and conditional branch instructions, for example *ifne, ifnull and if_icmpeq*.
- *Load and store operations*: Load a local variable onto stack, for example *aload*, *aload_0 and fload*. Store value from the operand stack into a local variable, for example *astore_0, astore_1*and *istore*.
- *Field access*: The value of an instance field may be retrieved with *getfield* and written with *putfield*. For static fields, there are *getstatic* and *putstatic* counterparts.
- *Method invocation*: Methods may either be called via static references with *invokestatic* or be bound virtually with the *invokevirtual* instruction. Super class methods and private methods are invoked with *invokespecial*.
- *Return Instruction*: The method return instructions, which are distinguished by return type, are *ireturn* (used to return values of type Boolean, byte, char, short, or int), *lreturn*, *freturn*, *dreturn* and *aretrun*. In addition, the *return* instruction is used to return from methods declared to be void.
- *Object allocation*: Class instance is allocated with the *new* instruction, array of basic type like int[] is allocated with *newarray*, array of references like String[][] is allocated with *anewarray* or *multianewarray*.
- *Conversion and type checking*: There are instructions for checking and converting basic types and instances. For example, *i2f* instruction is used to convert int to float. The *instanceof* instruction is used to determine if object is of given type.
- *Operand Stack management*: A number of instructions are provided for the direct manipulation of operand stack, for example *pop*, *dup* and *swap*.
- *Throwing Exception*: Exceptions are thrown using the *athrow* instruction.

A list of all instructions with detailed description can be found in the JVM Specification [13].

In Java, an *exception* is an event that occurs during the execution of a program that disrupts the normal flow of instructions. When an exception is raised,

control transfers to a block of instructions that can handle the exception. This block of instructions is called an *exception handler*. For example, the catch block in the left side of Figure 2.1 is the exception handler.

2.1.2 Software Testing

Software testing has been proved to be a valuable activity for determining whether a software system has faults. It is estimated that "testing consumes at least half of the labor expended to produce a working program" [5]. Beizer defines three distinct levels of testing, unit/component testing, integration testing and system testing. The objectives of each level are different. *Unit/Component testing* aims to show if the unit/component satisfies its functional specification and/or if its implemented structure matches the intended design structure. *Integration testing* aims to show inconsistencies between units or components. *System testing* concerns issues and behaviors that can only be exposed by testing the entire integrated system or a major part of it. There are two general testing approaches, *white-box* and *black-box*. *White-box* approaches, such as branch testing and path testing, require knowledge of the implementation based on the source code. *Black-box* approaches, such as functional testing and random testing, require knowledge of the specification details. Two approaches are complementary to each other.

Many researches on software testing have concentrated on the process of creating the set of test cases that is consisting of an input and expected output pairs [6, 7, 14]. A program is executed on the input and the expected output is compared to the actual output. Most of the researches on testing have revolved around the goal of selecting the small set of inputs that discover the large set of errors.

Test requirements are specific things that must be satisfied or covered during testing. For example, reaching every statement of a program is the requirements for statement coverage. A *testing criterion* is a rule or collection of rules that imposes the requirements on a set of test cases. Such as, data flow testing criteria require test cases to exercise certain paths based on data flow relationships. Testers measure the extent to

which a criterion is satisfied in terms of *coverage*, which is the percent of requirements that are satisfied [6]. There are lots of ways to classify adequacy criteria. One of the most common ways is by the source of information used to specify testing requirements and in the measurement of test adequacy. Therefore, an adequacy criterion can be specification-based and program-based.

A *specification-based* criterion specifies the required testing in terms of identified features of the specifications of the software, so that a test set is adequate if all the identified features have been fully exercised. A *program-based* criterion specifies testing requirements in terms of the program under test and decides if a test set is adequate according to whether the program has been thoroughly exercised. The criteria listed below are traditional program-based criteria.

- *Statement coverage* requires enough test cases to ensure that each statement in a program is executed at least once.
- *Decision coverage* (or *branch coverage*) requires enough test cases to ensure that each decision has a true and false outcome at least once.
- *Condition coverage* requires enough test cases to ensure that each condition in a decision takes on all possible outcomes at least once.

- *Data flow testing* [15, 16] tries to ensure that the correct values are stored into memory, and then that they are subsequently used correctly. A definition (*def*) is a statement where a variable's value is stored into memory. A *use* is a statement where a variable's value is accessed. A *definition-use pair* (or du-pair) of a variable is an ordered pair of a definition and a use, with the limitation that there must be an execution path from the definition to the use without any intervening redefinition of the variable. Data flow criteria requires tests to execute paths from specific definitions to uses. They select particular definition-use pairs to test. Two data flow testing criteria were first defined by Laski and Korel [15]. They proposed the *all-definitions* criterion, which requires that a test should cover a path from each definition to at least one use, and the ***all-uses*** criterion, which requires a test to cover a path from each def to all reachable uses.

Unfortunately, directly applying either the all-defs or the all-uses criterion to interprocedural testing is very expensive, both in terms of number of du-pairs and the difficulty of resolving the paths. Therefore, Jin and Offutt [17] proposed *coupling-based testing* (CBT) as an application of data flow testing to the integration level.

Coupling-based testing (CBT) [17] applies data flow testing to the integration level by requiring the program to execute data transfers from definitions of variable in a caller to uses of the corresponding variables in the callee unit. Instead of all variables definitions and uses, CBT is only concerned with definitions of variables that are transmitted just **before** calls (*last-defs*) and uses of variables just **after** calls (*first-uses*). The criteria are based on the following definitions:

- A *Coupling-def* is a statement that contains a last-def that can reach a first-use in another method on at least one execution path
- A *Coupling-use* is a statement that contains a first-use that can be reached by a last-def in another method on at least one execution path
- A *coupling path* is a path from a coupling-def to a coupling-use

Four levels of coupling-based integration test coverage criteria are defined between two units:

- *Call-coupling* requires the test cases to cover **all call-sites** of the called method in the caller method
- *All-coupling-defs* requires, for **each coupling-def** of a variable in the caller, the test cases cover at least one coupling path to **at least one reachable coupling-use**
- *All-coupling-uses* requires, for **each coupling-def** of a variable in the caller, the test cases must cover at least one coupling path to **each reachable coupling-use**
- *All-coupling-paths* requires the test cases to cover **all coupling paths** from each coupling-def of a variable to all reachable coupling-uses

2.1.3 Testability

Software testing is one of the most common ways to assure software quality and reliability, and is made easier by high software testability. Several different definitions of testability have been published [10, 18, 19, 20].

According to the 1990 IEEE standard glossary [18], testability is the "degree to which a component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met, and the degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met."

Voas and Miller [10] explained that testability enhances testing and claimed that increasing component testability is a primary key to improve the testability of component-based software. They define software testability by focusing on the "probability that a piece of software will fail on its next execution during testing (with a particular assumed input distribution) if the software includes a fault."

Binder [19] defined testability in term of *controllability* and *observability*. Controllability is the probability that users are able to *control* component's inputs (and internal state). Observability is the probability that users are able to *observe* component's outputs. If users cannot control the input, they cannot be sure what caused a given output. If users cannot observe the output of a component under test, they cannot be sure if the execution was correct.

Likewise, Freedman [20] considered testability based on the notions of controllability and observability. Controllability refers to the ease of producing all values of its specified output domain. Observability captures the degree to which a component can be observed to generate the correct output for a given input.

From above definitions, component testability generally refers to how easy to test. A high degree of testability indicates that any existing faults can be

revealed relatively easily during testing, inputs can easily be selected to satisfy some testing criteria and outputs of state variables can be observed during testing.

A component with good testability is important because test tasks are eased and test costs are reduced. There are various ways to improve testability. Gao, Tsao and Wu [21] introduced three basic approaches to increase software component testability: 1) *Framework-based testing facility*, 2) *Built-in tes*ts, and 3) *Automatic component wrapping for testing*. These three mechanisms construct testable components. *Framework-based testing facility* method, a well-defined framework (such as a class library) is developed to allow engineers to add program testing code into software components. *Built-in tests* method requires component developers to add test code and tests inside a software component to support self-checking and self-testing. *Automatic component wrapping for testing* method uses a systematic way to convert a software component into a testable component by wrapping it with the program code which facilitates software testing.

2.1.4 Mutation Testing

Mutation analysis [22] is often used to assess the adequacy of a test set. It is a fault-based testing strategy that starts with a program (or specification) to be tested and makes numerous small syntactic changes into the original program (or the specification). Program (or specification) with injected faults are called *mutants*. Mutants are obtained by applying *mutation operators* that introduce the simple changes into the original program (or specification). For example, changing relational operator:

The program P =

```
1. if ( x < 10 )
2.     doA( )
3. if ( x < 20 )
4.     doB( )
```

A mutation of P would be (line 1)

```
1. if ( x > 10 )
2.     doA( )
3. if ( x < 20 )
4.     doB( )
```

If a test set is capable of causing behavioral differences between original program (or specification) and mutant, mutant is considered as killed by the test. The product of the mutation analysis is a measure called mutation score, which indicates the percentage of mutants killed by a test set. If the mutation score is 100% or near, it indicates the adequacy of the test cases. Some mutants are functionally *equivalent* to the original program. Equivalent mutants always produce the same output as the original program, so cannot be killed by any test cases. Equivalent mutants are not counted in the mutation score.

2.2 Related Works

Several approaches relating to software testing and testability have been proposed in the literature. In this section we describe a number of them.

2.2.1 Testing Object-Oriented classes

Hong et al. [8] and Harrold and Rothermel [23] proposed methods that use information from developers to test object-oriented components. Hong et al. [8] proposed the method for a single-class testing. The method models the behavior of a single class as a finite state machine that is transformed representation into a data flow graph. This graph explicitly identifies the definitions and uses of each state variable of the class. Then conventional data flow testing can be applied to produce test case specifications that can be used to test the class.

Harrold and Rothermel [23] proposed a method for performing class testing by testing the data flow interactions in a class. Their approach consists of three levels of data flow testing, *intra-method testing*, *inter-method testing*, and *intra-class testing*. *Intra-method testing* has the same meaning as the unit testing of a procedure in procedural programs. *Inter-method testing* has the same meaning as the integrating testing. *Intra-class testing* performs testing on the interactions of public methods when they are called in random sequences. Their testing methods use the data flow relations from the program source to guide the selection of tests.

The above explained approaches [8, 23] relied on detailed analysis of the program source to generate test cases by considering all definitions and uses of variables from data flow analysis. Applying all definitions and uses of variables to generate test cases is very expensive, both in term of number of paths and the difficulty of resolving the paths. Our approach generates test cases based on data flow testing by considering only the first use and the last definition of variables. Our analysis method analyzes data flow information of a class-component at bytecode level because the source is not always available for software component.

2.2.2 Testability Analysis

A number of researches [10, 24, 25] have proposed the testability analysis approaches. McCabe [24] proposed McCabe metric to predict testability. This metric evaluates software complexity by measuring the cyclomatic number based on the number of execution paths in the control flow graph. This complexity measure is assumed to apprise the number of test cases in term of number of execution paths.

Voas et al. [10, 25] defined software testability as the probability that a piece of software will fail on its next execution during testing, provided it contains a fault. They believed that software failure only occurs when the following three necessary and sufficient conditions occur in the following sequence:

1. A input must cause a fault to be executed

2. Once the fault is executed, the succeeding data state must contain a data state error.
3. Once the data state error is created, the data state error must propagate to an output state.

From these three conditions, they defined *fault sensitivity* to analyze testability as multiplying the probabilities, named PIE analysis, that (1) the location containing the fault is executed, *execution*, (2) the fault corrupts the program's state, *inflection*, and (3) the corrupted state propagates to the output, *propagation*. High fault sensitivity indicates high testability and vice versa.

Our testability analysis technique is closely related to the mutation testing technique used by Voas et al. [10, 25]. Their work analyzes testability from three stages which are *execution*, *inflection* and *propagation*. They defined the *inflection* technique as making a fault to the source code and determining the program's state corruption as a result of the induced fault, and purposely defined the *propagation* technique as analyzing the corrupted state propagating to the output. Due to our assumption that the component's source codes are unavailable, the Voas et al's *inflection* technique is therefore not applicable. We induced faults to data states at the bytecode level.

CHAPTER III
CLASS-COMPONENT TESTABILITY

This chapter describes methodology and mechanism of a framework for class-component testability. Figure 3.1 shows an overview of the class-component testability process. First, class-component, which is bytecode instructions, is analyzed by the analysis tool called *Bytecode-based Class-Component Analysis*. This process extracts control flow and data flow information based on data flow analysis as described by part of subsection 2.1.2.

Results of the previous process are used to increase class-component testability in the *Increasing Class-Component Testability* process, and to measure class-component testability in the *Measuring Class-Component Testability* process. The details of each process are described in the subsequent sections.

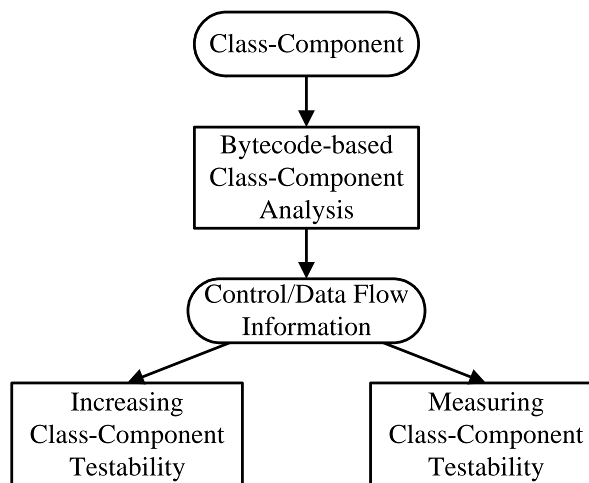

Figure 3.1: The Class-Component Testability Process

3.1 Bytecode-based Class-Component Analysis

This thesis is based on data flow testing that considers a flow between a variable definition and subsequent use of that variable. The supporting information for considering is control flow and data flow information.

The most common method to represent the control flow information is a *Control Flow Graph* (*CFG*) [1], which was originally proposed for compiler optimization. Each node in a CFG represents a statement or a basic block of statements, and edges represent the flow of control between nodes. To represent both control and data flow information, CFG is extended by collecting variables defined and used in each node. The extended CFG to represent control and data flow information is called *Control-Data Flow Graph* (*CDFG*).

Conventional program analysis collects control and data flow information from source code of the application program. Because of the lack of source code due to inaccessibility, our analysis technique gathers such information at the *bytecode* level.

Our bytecode analysis results in a CDFG for each method of a class-component. The CDFG includes both control and data flow information. The CDFG also includes exception handling control, which was not defined for the CFG. The exception handling control flow can be raised in the class-component through a *throw* statement (for example, *throw* new ArithmeticException()). The CDFG construction process is as follows. First, the bytecode instructions are extracted and partitioned into *basic blocks*, and then the flows of control and data are added. The details are described in the following sections. Table 3.1 shows bytecode instructions used in the CDFG construction process.

Table 3.1: Bytecode Instructions Used in the CDFG Construction Process

Group	Bytecode Instructions
Unconditional branch	goto
Conditional branch	if_acmpeq, if_acmpne, if_icmpeq, if_icmpge, if_icmpgt, if_icmple, if_icmplt, if_icmpne, ifeq, ifge, ifgt, ifle, iflt, ifne, ifnonnull, ifnull, lookupswitch, tableswitch
Return	areturn, dreturn, freturn, ireturn, lreturn, return
Athrow	athrow

3.1.1 Determining Basic Blocks

A *basic block* is a sequence of consecutive bytecode instructions in which flow of control enters at the beginning and leaves at the end without halting or branching except at the end. In a basic block, if any instruction is executed, all instructions will be executed. A basic block has only one entry point and one exit point. To create a basic block, *leader instructions,* instructions that begin basic block, are identified. A leader instruction can be:

- The first instruction,
- Any instruction that is the target of either a conditional branch group instruction or an unconditional one,
- Any instruction that immediately follows either a conditional branch group instruction or an unconditional one,
- Any instruction that immediately follows a return group instruction or *athrow* instruction (exception handling), and
- The first instruction of an *exception handler*.

After identifying a leader instruction, a basic block is defined as consisting of a leader and all instructions up to but not including the next leader. An *EXIT* block is added to be the exit point. For example, the bytecode instructions for method *calCoeff* in Figure 2.1 are divided into the following basic blocks: [0-4], [7-14], [15-21], [24-29], [32-35], [38], [41-47] and [50] as shown in Figure 3.2. Each basic block is analyzed to gather the definition and use information (*DefUse information*) for data flow analysis. DefUse information refers to variable definitions and uses. An instruction is considered to perform a *definition* of a variable (*Def instruction*) if a value is stored into that variable from the operand stack. An instruction is considered to perform a *use* of a variable (*Use instruction*) if its value is accessed and loaded onto the stack. In Java bytecode, there are two types of field variables, *instance fields* (non-static fields) and *class fields* (static fields). The instance fields are unique to each object of the class. The class fields are unique to the entire class. This thesis focuses on instance fields which are defined and used between methods, therefore local variables will not be considered.

```
0:    aload_0           //BasicBlock0
1:    getfield          Coeff.rate I (2)  //use rate
4:    ifne              #15 //branch to address 15
------------------------------------------------------------------
                        //BasicBlock1
7:    new               <java.lang.ArithmeticException> (3)
10:   dup
11:   invokespecial     java.lang.ArithmeticException.<init> ()V (4)
14:   athrow            //Throwing exception
------------------------------------------------------------------
15:   aload_0           //BasicBlock2
16:   getfield          Coeff.rate I (2)  //use rate
19:   bipush            20
21:   if_icmpge         #32 //branch to address 32
------------------------------------------------------------------
24:   aload_0           //BasicBlock3
25:   iconst_5
26:   putfield          Coeff.co I (5)  //define co
29:   goto              #38 //branch to address 38
------------------------------------------------------------------
32:   aload_0           //BasicBlock4
33:   bipush            15
35:   putfield          Coeff.co I (5)  //define co
------------------------------------------------------------------
                        //BasicBlock5
38:   goto              #50 //branch to address 50
------------------------------------------------------------------
                        //BasicBlock6
41:   astore_1          //Exception handle block
42:   getstatic         java.lang.System.out Ljava/io/PrintStream; (6)
45:   ldc               "Catch Exception" (7)
47:   invokevirtual     java.io.PrintStream.println (Ljava/lang/String;)V (8)
------------------------------------------------------------------
50:   return            //BasicBlock7
```

Figure 3.2: Partitioning Method calCoeff in Figure 2.1 into Basic Blocks

Because of the complexity of array bytecode instructions, we only focus on an array of type *int*. Furthermore, the DefUse information of any element within array is considered the DefUse information of the whole array. The reference Java bytecode instructions used to gather the DefUse information is shown in Table 3.2.

Table 3.2: DefUse Information

Bytecode Instructions	**Description**	**Def/Use Instruction**
getfield	Load a value of an instance field onto the operand stack	Use
putfield	Store a value from the operand stack as an instance field	Def
iaload	Load an array component onto the operand stack	Use
iastore	Store a value from the operand stack as an array component	Def

3.1.2 Constructing the CDFGs

After each basic block has been defined, edges associated with the flow of control are added. An edge is added from basic block B1 to B2, depending on the type of the last instruction in B1, which could be one of the following cases:

Case1: An instruction of unconditional branch group: An edge is added from B1 to the basic block whose leader is the target of the branch instruction of B1 (e.g. from *BasicBlock3* to *BasicBlock5* in Figure 3.3, *Flow7* in Figure 3.3).

Case2: An instruction of conditional branch group: Two edges are added from B1. The first is to the basic block whose leader is the first instruction that directly follows the last instruction of B1 (e.g. from *BasicBlock0* to *BasicBlock1*, *Flow1*). The second is to the basic block whose leader is the target of the branch instruction of B1 (e.g. from *BasicBlock0* to *BasicBlock2*, *Flow2*).

Case3: An instruction of return group: An edge is added from B1 to the exit point, *EXIT* (e.g. from *BasicBlock7* to the *EXIT* block, *Flow10*).

Case4: An *athrow* instruction: An edge is added from B1 to the basic block whose leader is the first instruction of the associated exception handler. If there is no associated exception handler, an edge is added to the exit, *EXIT* (e.g. from *BasicBlock1* to *BasicBlock6*, *Flow5*).

Case5: Not an instruction of branch, return or athrow group: This happens when a B1 ends just before a leader of another basic block. Add an edge from B1 to the next basic block (e.g. from *BasicBlock4* to *BasicBlock5*, *Flow6*).

Figure 3.3 shows the flows between basic blocks of method calCoeff.

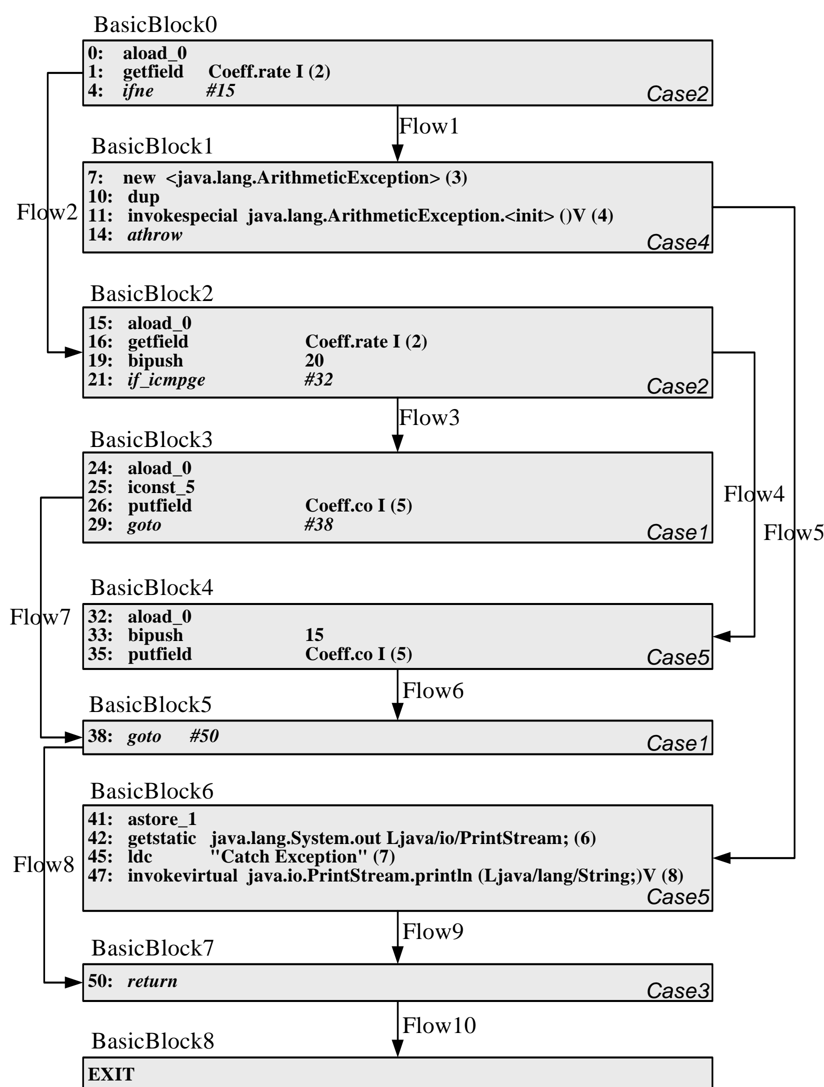

Figure 3.3: The Flows between Basic Blocks of Method calCoeff

We illustrate our technique with a vending machine example taken from Harrold et al. [26]. The Java source code and bytecode instructions for the vending machine are shown in Figures 3.4 and 3.5 respectively. The numbers preceding the line in Figure 3.5 indicate the instruction positions of the bytecode. The "…" indicates omitted instructions. Figure 3.6 depicts the CDFGs of each method of vending machine. In this figure, a basic block is represented as the compartmentalized rectangle. The top compartment contains the basic block number, the middle compartment contains the first and last instruction positions of the basic block, and the bottom compartment contains the sequence of DefUse information. Each element of this sequence contains *d* (a definition) or *u* (a use), the variable name, and the position of the instruction. For example, the element (*d, Type, 16*) in basic black 0 of method <*init*> indicates that the variable *Type* is defined (*putfield* instruction) at position 16. A directed arrow shows the flow of control between basic blocks. A *predecessor block* of a current block is the basic block that has the flows of control to the current block. A *successor block* of current block is the basic block that has the incoming flows of control from the current block.

Data flow analysis is a technique for obtaining variables' relationships from flow graphs. This technique examines definitions and the subsequent uses of variables. Suppose instruction I_1 defines a value to *x*, which instruction I_2 then uses. Then, instructions I_1 and I_2 have a data flow relationship. Using DefUse information from the previous step, data flow analysis can be processed by traversing the basic blocks in the CDFGs. For the vending machine example, the data flow relationship of variable *Type* within method *vend* is in basic block 0 at position 7 (Def) and basic block 2 at position 29 (Use). Data flow relationships can also be obtained between methods, for example, variable *curQtr* is defined in basic block 0 of method <*init*> and then used in basic block 0 of method *addQtr*.

```
1   class VendingMachine {
2
3      private int total = 0;
4      private int curQtr = 0;
5      private int Type = 0;
6      private int availType = 2;
7
8      void addQtr() {
9         curQtr = curQtr + 1;
10     }
11
12     void returnQtr() {
13        curQtr = 0;
14     }
15
16     void vend ( int selection ) {
17        int MAXSEL = 20;
18        int VAL        = 2;
19        Type          = selection;
20        if ( curQtr == 0 )
21            System.err.println ("No coins inserted");
22        else if ( Type > MAXSEL )
23            System.err.println ("Wrong selection ");
24        else if ( !available( ) )
25            System.err.println ("Selection  unavailable");
26        else {
27            if ( curQtr < VAL )
28                    System.err.println ("Not enough coins");
29            else {
30                    System.err.println ("Take selection");
31                    total     = total+ VAL;
32                    curQtr = curQtr - VAL;
33            }
34        }
35        System.out.println ("Current value = " + curQtr );
36     }
37
38     boolean available( ) {
39         if (availType == Type)
40              return true;
41        else
              return false;
42     }
43   } // class VendingMachine
```

Figure 3.4: The Vending Machine Class

```
void <init>()
0:   aload_0
…
6:   putfield        VendingMachine.total I (2)
…
11:  putfield        VendingMachine.curQtr I (3)
…
16:  putfield        VendingMachine.Type I (4)
…
21:  putfield        VendingMachine.availType I (5)
24:  return

void addQtr()
0:   aload_0
1:   aload_0
2:   getfield        VendingMachine.curQtr I (3)
…
7:   putfield        VendingMachine.curQtr I (3)
10:  return

void returnQtr()
0:   aload_0
1:   iconst_0
2:   putfield        VendingMachine.curQtr I (3)
5:   return

void vend(int arg1)
0:   bipush          20
…
7:   putfield        VendingMachine.Type I (4)
10:  aload_0
11:  getfield        VendingMachine.curQtr I (3)
14:  ifne            #28
17:  getstatic       java.lang.System.out
…
25:  goto            #112
28:  aload_0
29:  getfield        VendingMachine.Type I (4)
32:  iload_2
33:  if_icmple       #47
36:  getstatic       java.lang.System.out
…
44:  goto            #112
47:  aload_0
48:  invokevirtual   VendingMachine.available ()Z (10)
51:  ifne            #65
54:  getstatic       java.lang.System.out
…
62:  goto            #112
65:  aload_0
66:  getfield        VendingMachine.curQtr I (3)
69:  iload_3
70:  if_icmpge       #84
73:  getstatic       java.lang.System.
…
81:  goto            #112
84:  getstatic       java.lang.System.out
…
94:  getfield        VendingMachine.total I (2)
…
99:  putfield        VendingMachine.total I (2)
…
104: getfield        VendingMachine.curQtr I (3)
…
109: putfield        VendingMachine.curQtr I (3)
112: getstatic       java.lang.System.out
…
128: getfield        VendingMachine.curQtr I (3)
…
140: return

boolean available()
0:   aload_0
1:   getfield        VendingMachine.availType I (5)
4:   aload_0
5:   getfield        VendingMachine.Type I (4)
8:   if_icmpne       #13
11:  iconst_1
12:  ireturn
13:  iconst_0
33:  ireturn
```

Figure 3.5: The Bytecode Instructions of Vending Machine

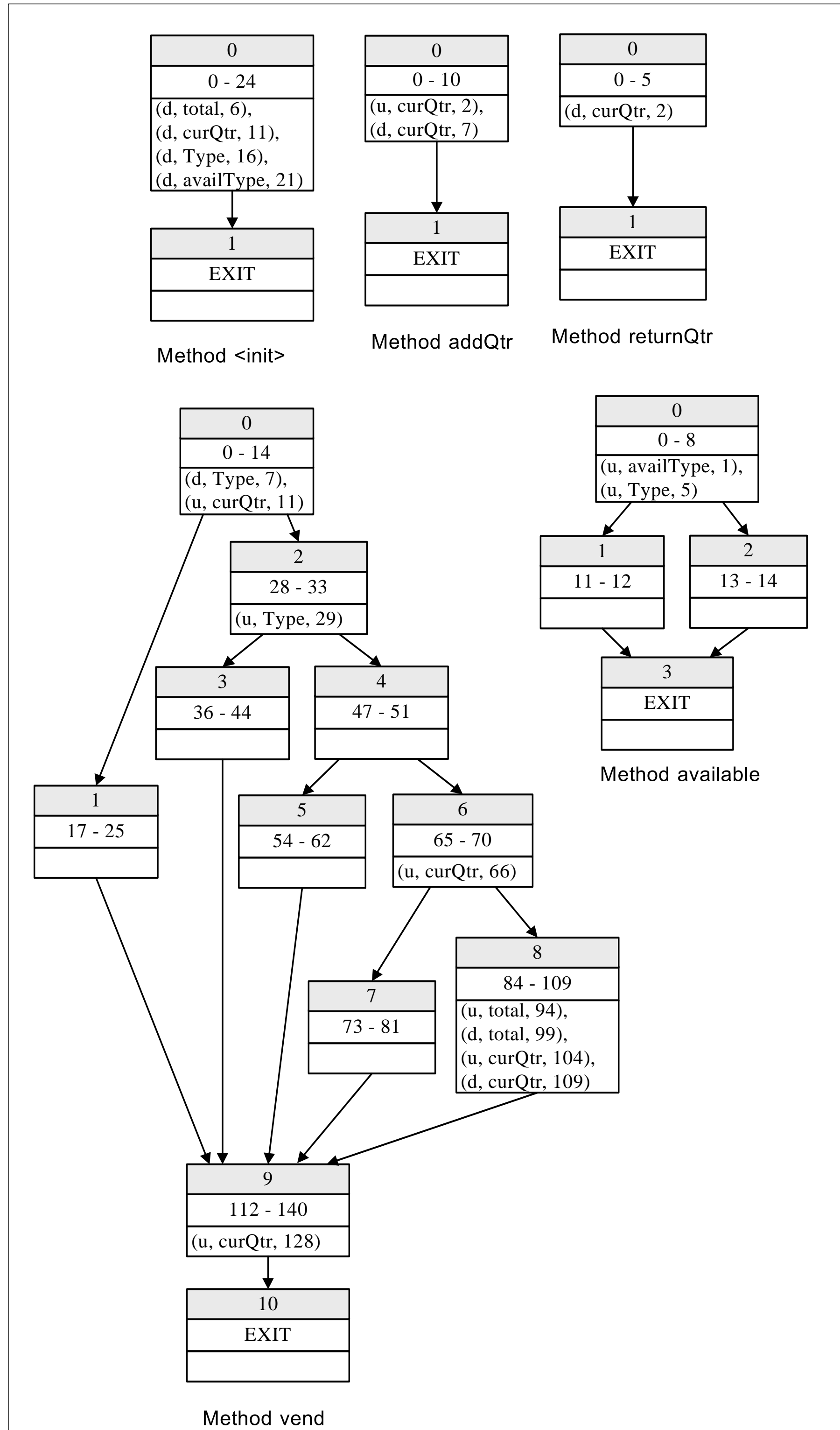

Figure 3.6: The CDFGs of Each Method of Vending Machine

3.1.3 Applying the CDFGs

An analysis tool was implemented for automatically analyzing java .class file and generating CDFGs. The tool was implemented by using an open source tool from Apache/Jakarta [12], a bytecode manipulation library called BCEL (Byte Code Engineering Library). The BCEL API can help to analyze, create and manipulate Java bytecode files.

Figure 3.7 illustrates the class-component analysis process. The process consists of two major components: (1) the *Class Parser* and (2) the *Control-Data Flow Graph Generator* (*CDFGen*). The *Class Parser* parses the java .class file and creates the *JavaClass* object, which represents all the information about the class (constant pool, fields, methods etc.). The *CDFGen* consists of three parts: the *leader generator*, the *basic block generator* and the *flow generator*. The *leader generator* generates the *Leader Hashtable* that will be used by the *basic block generator* and the *flow generator*. The *Leader Hashtable* contains leaders that were mentioned in section 3.1.1. The *basic block generator* divides bytecode instructions into a collection of basic blocks. The *Def* instruction and *Use* instruction are also collected for each basic block. The *flow generator* generates flows of control between basic blocks. When the three processes of *CDFGen* finish, a CDFG has been created for a method.

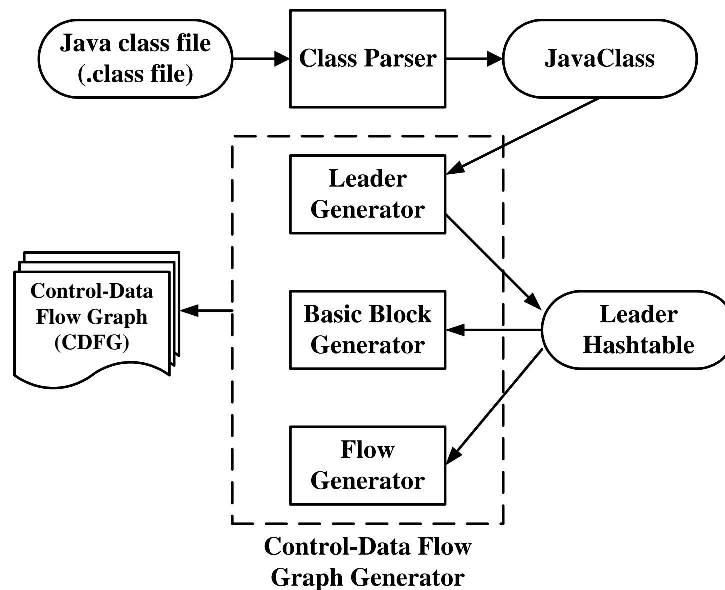

Figure 3.7: Class-Component Analysis Process

3.1.4 Definition-Use of Method (DUM)

The previous process constructs the *Control-Data Flow Graph* (*CDFG*) of a method to model the flow of control and data through that method. This process uses the CDFGs of a class-component to collect the definition and use information. The process collects (1) the definition and use information for all variables of each method and (2) the first use and the last definition information. The information gathered for a method is collectively called the *Definition-Uses of Method* (*DUM*). To collect definitions and uses information for all variables of a method, the associated CDFG of a method is traversed from the first basic block (basic block 0) to the last basic block (*EXIT* block). To collect the first uses, the CDFG is traversed by starting from the first basic block and then following with each successor block, in a depth first manner. To collect the last definitions, the CDFG is traversed by starting from the last basic block and then following backward through predecessor blocks, again in a depth first manner. Table 3.3 shows the DUMs of vending machine. Columns Use and Def show the sequence of use and definition instruction positions of variables. Column First-Use and Last-Def show the sequence of first use and last definition instruction positions of variables.

Table 3.3: The DUMs of Vending Machine

Method Name	Variable Name	Positions			
		Use	Def	First-Use	Last-Def
<init> ( )	curQtr		11		11
	total		6		6
	Type		16		16
	availType		21		21
addQtr ( )	curQtr	2	7	2	7
returnQtr ( )	curQtr		2		2
Vend ( )	curQtr	11, 66, 104, 128	109	11	109
	total	94	99	94	99
	Type	29	7	29	7
available ( )	Type	5		5	
	availType	1		1	

As shown in Figure 3.8, the *Definition-use Generator* was implemented to generate the DUMs for each method. The next section explains how the DUMs are used to increase testability, controllability and observability, of a class-component.

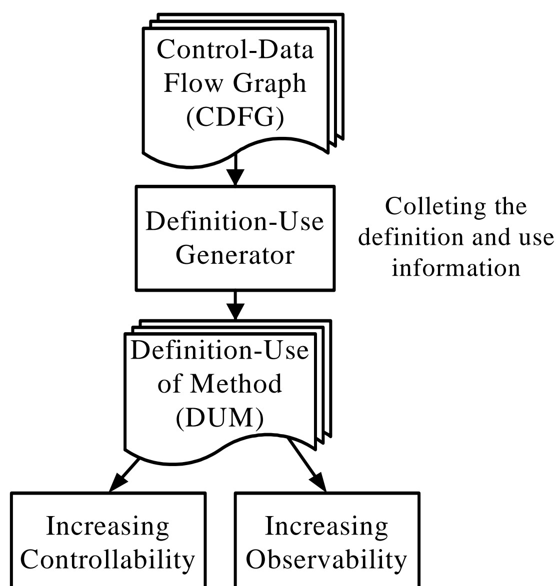

Figure 3.8: The Process of Collecting the Definition and Use Information

3.2 Increasing Class-Component Testability

Testing software is easier when testability is high, and, in general, increasing testability makes detecting faults easier. This section explains the process used to increase class-component testability. The DUMs from the previous step are used to increase testability that is proportionally influenced by: *controllability* and *observability*.

3.2.1 Increasing Controllability

Binder [19] defined the significance of controllability as "if user cannot control the inputs, they cannot be sure what caused a given output". Controllability focuses on the ease of controlling component's inputs. This means that a class-component that supports various ways of supplying inputs to exercise the class-component as necessary tends to provide better controllability.

To increase controllability, the DUMs mentioned in previous step are used to collect definition-use pairs of a variable between the last definitions and the first uses. The definition-use pairs of a variable are called *Definition-Use Couplings for Testing (DUCoT)*. For example, the variable *total* of the vending machine in Table 3.3, the last definitions are in method *<init>* at position 6 and method *vend* at position 99, and the first use is in method *vend* at position 94. The remainder of this subsection refers to the last definition at position *x* and the first use at the position *y* as the *last-def location x* and the *first-use location y*. A *DUCoT* is defined as follows:

Definition The *DUCoT* of variable *v* is a tuple, *DUCoT* (*v*) = (D_L, U_F)

- D_L is a finite set of last definitions of variable *v*

  Each element of D_L is $M_d[L_d]$ where

  M_d is a method that defines variable *v*, and

  L_d is a last-def location in M_d where variable *v* is defined

- U_F is a finite set of first uses of variable *v*

  Each element of U_F is $M_u[L_u]$ where

  M_u is a method that uses a variable *v*, and

L_u is a first-use location in M_u where variable *v* is used

Figure 3.9 shows the DUCoTs for variables of vending machine. From the figure, the first DUCoT is for variable curQtr, DUCoT(curQtr) = (D_L, U_F). D_L is a set that consists of four elements, *<init>[11]*, *addQtr[7]*, *returnQtr[2]* and *vend[109]*. Each element contains two parts, M_d and [L_d]. The first part, M_d, is a method's name. L_d in the second part is a last-def location in method M_d. For example, the last element of D_L, vend[109], indicates a method *vend* and a last-def location 109. U_F is a set that consists of two elements, *addQtr[2]* and *vend[11]*. Each element contains two parts, M_u and [L_u]. The first part, M_u, is a method's name. L_u in the second part is a first-use location in method M_u. For example, the last element of U_F, vend[11], indicates a method *vend* and a first-use location 11.

DUCoT (curQtr) = ({<init>[11], addQtr[7], returnQtr[2], vend[109]},
{addQtr[2], vend[11]})

DUCoT (total) = ({<init>[6], vend[99]}, {vend[94]})

DUCoT (Type) = ({<init>[16], vend[7]},
{vend[29], available([5]})

DUCoT (availType) = ({<init>[21]}, {available[1]})

Figure 3.9: The DUCoTs of Vending Machine's Variables

The DUCoTs are used to increase controllability by supporting test case generation to cover all the necessary tests. These test cases are generated according to the coupling-based testing criteria proposed by Jin and Offutt [17], as described in Section 2.1.2. The coupling-based testing criteria are a collection of rules that imposes requirements on a set of test cases. Applying coupling-based testing to component testing requires some minor modifications to the terminology.

The *All-coupling-uses* criterion requires that for each **coupling-def**, at least one test case executes a path from the def to each reachable **coupling-use.** A coupling-def is an instruction that contains a last-def that can reach a first-use in another

method on at least one execution path. A coupling-use is an instruction that contains a first-use that can be reached by a last-def in another method on at least one execution path. An adaptation of the standard *All-coupling-uses* for a component by using DUCoT is given in the following definition.

Definition Let (D_L, U_F) be a DUCoT of variable *v*. The *All-coupling-uses* of variable *v* is defined as

$$AllCoU(v) = (D_L \times U_F) = \{ (M_d[L_d], M_u[L_u]) \mid \forall M_d[L_d] \in D_L \text{ and } \forall M_u[L_u] \in U_F) \text{ and } M_d = M_u \rightarrow L_d > L_u \}$$

In this thesis, we identify *test requirements* in form of definition-use pairs that are to be tested. The test requirement for an ordered definition-use pair (M_d [L_d], M_u [L_u]) of variable *v* requires the path to execute from the last-def location L_d of method M_d to the first-use location L_u of method M_u without any intervening redefinitions of the variable *v*. The *All-coupling-uses* considers definition-use pair between methods, a last-def in a method and a first-use in another method. Therefore, if the first-use and last-def locations are in the same method, the first-use location must be appeared before the last-def location. Example of the test requirements of All-coupling-uses of vending machine are shown in Table 3.4.

Table 3.4: The Test Requirements of All-coupling-uses of vending machine

#	Variable	All-coupling-uses	#	Variable	All-coupling-uses
1	curQtr	<init> [11], addQtr [2]	8		vend [109], vend [11]
2		<init> [11], vend [11]	9	total	<init> [6], vend [94]
3		addQtr [7], addQtr [2]	10		vend [99], vend [94]
4		addQtr [7], vend [11]	11	Type	<init> [16], vend [29]
5		returnQtr [2], addQtr [2]	12		<init> [16], available [5]
6		returnQtr [2], vend [11]	13		vend [7], available [5]
7		vend [109], addQtr [2]	14	availType	<init> [21], available [1]

A tool to automatically generate DUCoTs and test requirements for a class-component was implemented. The tool provides two main modules, the generation of first-use and last-def of variables, and of All-coupling-uses test requirements as shown in Figure 3.10. The *First-use and Last-def Generator* uses DUMs to generate

DUCoTs. The *Coupling-based Test Generator* uses DUCoTs to generate test requirements of All-coupling-uses.

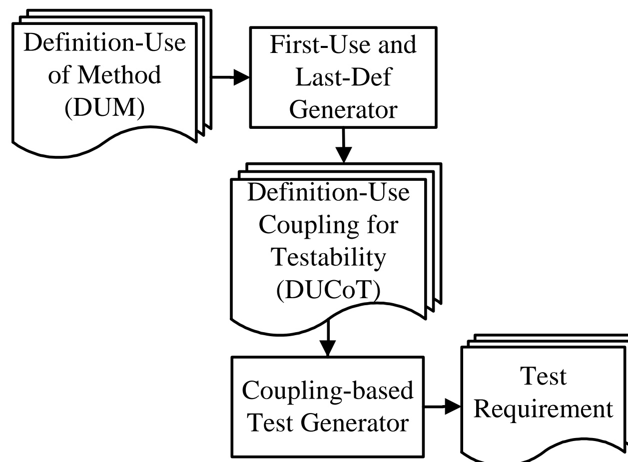

Figure 3.10: Test Requirement Generation Process

According to the test requirements generated, test cases are then created to satisfy these requirements. A test case is a sequence of method calls. For example, test requirement for a variable, (M_d [L_d], M_u [L_u]), causes the execution to reach two specific locations. The first is the last-def location L_d of method M_d, which defines that variable (*required def location*). The second is the first-use location L_u of method M_u, which uses that same variable (*required use location*). Moreover, this execution must not redefine the variable between these two locations. That is, this must be a *def-clear path execution*. Some of test requirements are infeasible because the required locations cannot be reached or the execution does not produce a def-clear path. To eliminate infeasible paths during test case generation process, instrumentation is developed.

An overall process of test case generation is shown in Figure 3.11. The *Definition-Use Track Instrumentation* was implemented to automatically instrument java .class file and to create the *Instrumented Class*. The instrumentation is performed at bytecode level by inserting auxiliary instructions at all definition and use locations. The definition and use locations are indicated by DUMs defined in section 3.1.4. The inserted instructions are used to record the reached locations. The *Temporary Test Data*

is first generated by the *Initial Test Data Generator* module according to a test requirement. The *Execution and Evaluation* executes the *Instrumented Class* against the *Temporary Test Data* and results the sequence of reached locations. The *Execution and Evaluation* also automatically checks the required def and use locations and checks that the execution is a def-clear path execution by using the sequence of reached locations. If the execution is not def-clear path or the required locations are not reached, the *Test Data Modifier* modifies the *Temporary Test Data* by adding a method call or by changing the values of the method call parameters. The initialization and modification of the test data are done manually. For example, in Table 3.4, the fourth test requirement, (addQtr[7], vend[11]), requires the execution to reach the location 7 of method *addQtr* and the location 11 of method *vend*. The initial test data for this test requirement consists of the vending machine class object, the call to method *addQtr*, and the call to method *vend*. The parameter of method *vend* is randomly generated, e.g. 1. This initial test data – *new(), addQtr(), vend(1)* – when executed, reaches the location 7 of method *addQtr* and location 11 of method *vend*, and creates the def-clear path execution. Therefore, the test case for this test requirement is *new(), addQtr(), vend(1)*. From the same table, considering the ninth test requirement, (<init>[6], vend[94]), which requires the execution to reach the location 6 of method *<init>* and the location 94 of method *vend*. The initial call, <init>, is called automatically by the system when a new class object, new(), is created. So the test data is initialized, using the same parameter as before, to be *new(), vend(1)*. When such test data is executed, the location 94 of method *vend* could not be reached. To correct this, the test data is modified by adding the calls to method *addQtr* and changing the parameter value of method *vend* to 2. As a result, the modified test case of the ninth test requirement in Table 3.4 is *new(), addQtr(), addQtr(), vend(2)*. The automated process of initialization and modification of test data is a complex problem, and, therefore, is left as part of the future work.

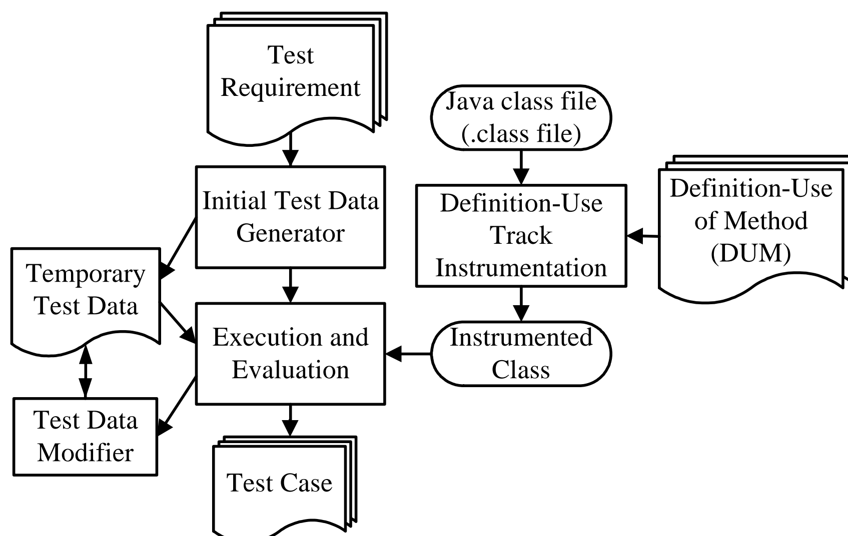

Figure 3.11: Test Case Generation Process

The step by step to summarize the test case generation process for a test requirement for a variable with parameters, location L_d of method M_d and location L_u of method M_u ($M_d[L_d]$, $M_u[L_u]$), is illustrated as follows:

1. First, an initial test data is generated with first creating a class-component object, the making a method call to M_d and later a method call to M_u. Note that, the initial test data, specifically method parameters, is randomly generated to fulfill requirement. This data is assigned as "temporary test data."
2. Next, the instrumented class is executed with the temporary test data. At the end of the execution, the sequence of the reached locations is recorded.
3. After that, the recorded reached locations are evaluated. If (1) the sequence of reached locations executed contains the required def location L_d in method M_d, (2) the sequence of locations executed contains the required use location L_u in method M_u after L_d in method M_d, and (3) there is no other definition of the variable between location L_d and location L_u, the temporary test data finalized and established as our "Test Case". Otherwise, the temporary test data is modified by adding a method call or by changing the values of the method call arguments, and repeat step 2.

The result of test case generation for a test requirement is the sequence of creating a class-component object and making method calls that cause execution of the required def location and the required use location with a def-clear path.

3.2.2 Increasing Observability

Binder [19] makes the following point about observability in object-oriented software: "if users cannot observe the output, they cannot be sure how a given input has been processed." Observability focuses on the ease of observing outputs. Observability requires that test engineers be able to determine if the software behaves correctly during testing. For black-box software, its observability is inherently limited because only the outputs are visible. Due to the encapsulation and data hiding properties of such black-box software, despite their direct benefits, cause problems with testing by making object-oriented software less observable [10]. This is because the internal state is not readily available. This makes it possible for the internal state to be erroneous while still yielding correct outputs. Hence, being able to access the internal state is a crucial task for testing.

An approach at accessing the internal state is by way of debugging. However, a debugger generally provides a view of all the state information at a certain point during an execution. The larger the size of a program, the less the observability it is. To alleviate this problem, we derived a method for specifying particular observation points to observe internal states of variables during testing. Typically, internal states of classes are not immediately and directly available to test engineers. For example, from Figure 3.4, the state variable *total*, as defined in line 31, is a private variable. So the client of the vending machine class consisting of this private variable (*total*) would not be able to access directly. Suppose that line 31 has a fault, no test case could detect it. To deal with this problem, being able to observe intermediate values of state variables during testing by using temporary variables is necessary.

Therefore, in this thesis, a technique called **observability probes** is introduced. This observability probes keep track of relevant internal state variables at

their definition and use locations, and assert the correctness of such state variables during testing. Such instrumentation is used to help the observability probes process.

The intermediate values of state variables can be observed by inserting auxiliary instructions at bytecode level. Such instructions are inserted into the class-components in such a way that the overall program behavior would not change. The instrumented instructions are classified into three groups. The first group defines instructions for creating temporary variables called *tracking variabl*e for each variable of a class-component. These variables are defined as public so they are accessible by other test classes. For example, variables *d_curQtr* and *u_curQtr* are created to be tracking variables of variable *curQtr* for storing its values at its definition location and its use location, respectively. The second group defines instructions for storing the values of each variable into its corresponding tracking variable created from the first group instructions. For example, the variable *d_curQtr* is assigned the value of the variable *curQtr* at its definition location. Similarly, the variable *u_curQtr* is assigned the value of the variable *curQtr* at its use location. The second group instructions are inserted at every location of the variable definition and every location of the variable use.

The last group is instructions used for asserting the correctness of the values of the internal state variables. The assertion is performed at every use location. The assertion is to see if the temporarily stored value at the use location is equal to the temporarily stored value of the corresponding state variable at the definition location, i.e., *u_curQtr* = *d_curQtr*. If the two values associated with the state variable are equal, that variable retains its right value. Otherwise, the variable has been unexpected redefined. Referring to table 3.4, which provides all test requirements of all relevant state variables for the vending machine class example. For example, to test the *curQtr* variable, all test requirements defined by ordered pairs from line 1 to line 8 must be executed. Specifically, the first test requirement (<*init*>[11], addQtr[2]) is first translated to the test case *new(), addQtr()*, as depicted in Figure 3.11. The *new()* method makes a call to the method <*init*>. When this test case is executed, the respective definition location, line 11 in Figure 3.5, of the method <*init*> and the respective use location, line

2 in Figure 3.5, of the method *addQtr* have been reached. At the point of the definition and use locations, the state variables of variable *curQtr* are stored into the corresponding tracking variables, *d_curQtr* and *u_curQtr*. At the point of the use location, an assertion is performed to compare the value of the tracking variable at its use location (*u_curQtr*) to the value of the tracking variable at its definition location (*d_curQtr*).

The mentioned observability probes process has been implemented as shown in Figure 3.12. A java .class file to be tested is instrumented by the *Observable Instrumentation* to add the three groups of instructions as previously stated. The *Observable Class* is produced by referring to the definition and use locations of the DUMs, described in section 3.1.4.

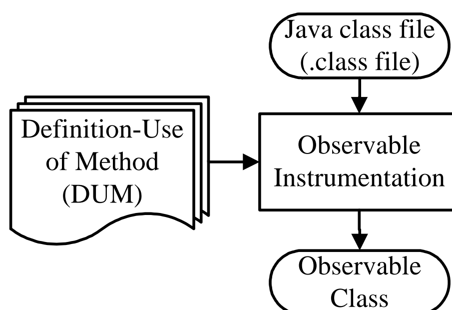

Figure 3.12: Observability Probes Process

Figure 3.13 shows the overall process of increasing class-component testability. From the figure, class-component (.class file) is analyzed at bytecode level by the *Betycode-based Class-component Analysis* process. This produces the *Definition-Use of Method* (DUM) for each method of the class-component being tested. The generated DUMs are then used in the *Increasing Controllability* process to generate test cases and used in the *Increasing Observability* process to monitor internal state variables. The *Increasing Controllability* process generates test cases according to All-coupling-uses criteria for the class-component being tested. The *Increasing Observability* process instruments a class-component to produce the *Observable Class*. When the *Observable Class* is executed by the *Execution and Assertion* module against

test cases, the execution results include the normal program outputs and the internal state variable assertion.

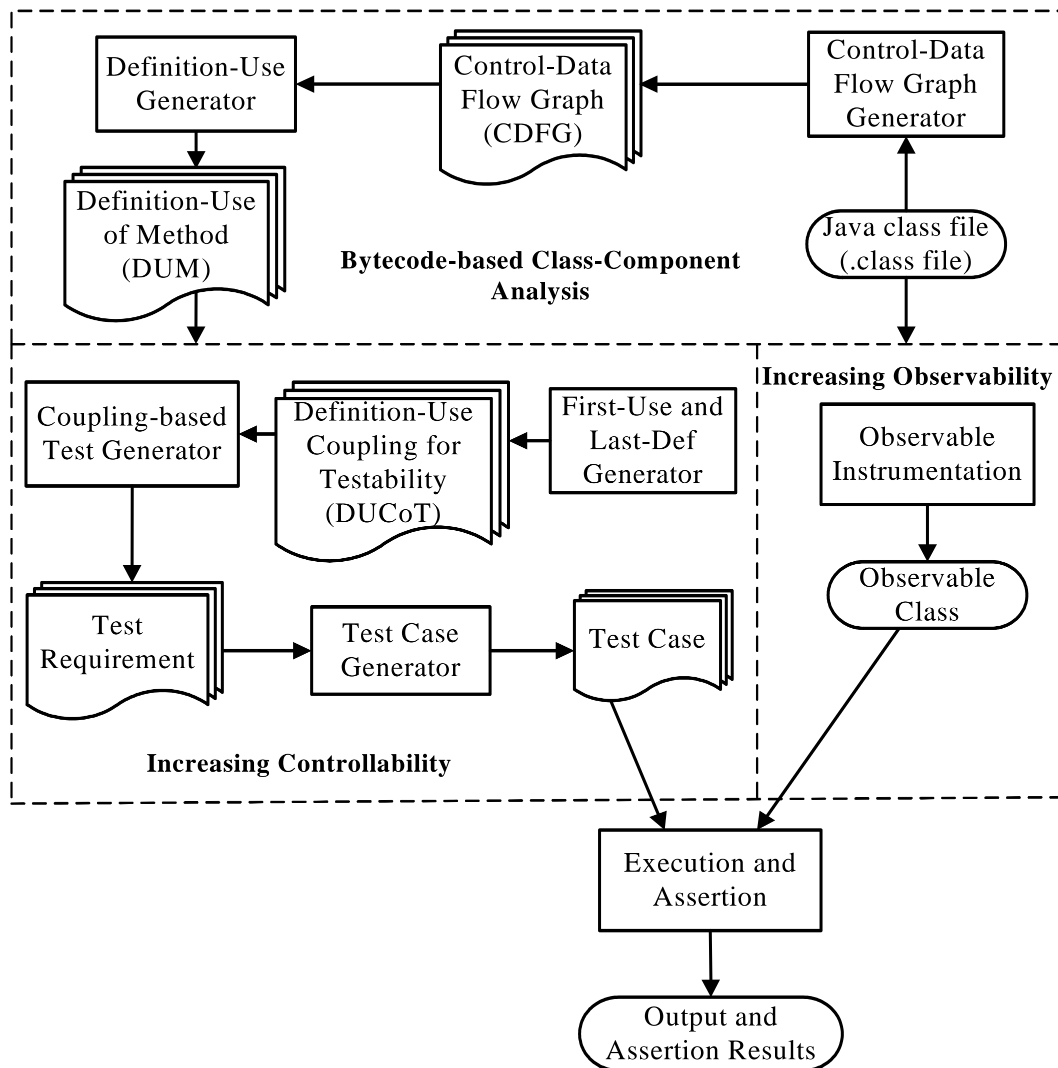

Figure 3.13: The Process of Increasing Class-Component Testability

3.3 Measuring Class-Component Testability

The goal of this section is to find a measurement that can be used to assess a component's testability. Generally, there exist two general testability measurement approaches. The first approach focuses on predicting the effort needed for program testing [24]. For example, if a particular software has high complexity in terms of the number of flows, much effort may be needed to satisfy a test criterion, thus reducing the software testability. The second approach focuses on the fault revealing ability during testing [10, 25], resulting in high degree of testability. This thesis adopted the concept of the second approach to measure a component's testability. In this approach, the testability measurement is defined as the probability that existing faults are revealed during testing. A fault can be revealed when a program segment that contains the fault is executed and the fault affects the output. Consequently, testability can be measured by the product of the probability that faults are executed (execution probability) and the probability that the executed faults propagate to the output (propagation probability). A location of variable definition and use is a good place to look for error [19]. Thus, the locations of variable definition and use will be used to measure testability.

3.3.1 Execution Analysis

Execution analysis executes a class-component and records the locations executed by each test case. The execution probability of a location l, $E(l)$, is estimated to be the percentage of test cases that execute that location. The algorithm for finding an execution probability of a particular location is as follows:

1. Assign a location number to each definition and each use location of a class-component.
2. Initialize an array *counter* to zeroes, where the size of *counter* is the number of definition and use locations in a class-component.
3. Execute a class-component by using a test case. If a location l is executed, the element of array *counter* that corresponds to the location l is increased the value by one.

4. Repeat algorithm step 3 *m* times with different test cases.
5. Divide each element of *counter* by *m* to calculate an execution probability of each location. For example, the execution of a location *l* is estimated to be *counter*[*l*]. The execution probability of location *l* is *counter[l]/m*.

The process of execution analysis was implemented as illustrated in Figure 3.14. From the figure, there are two major components: the *Execution Instrumentation* and the *Execution Analysis*. The *Execution Instrumentation* instruments java .class file (original class) and automatically creates the *Instrumented Class* as inputs for the *Execution Analysis* component. The instrumentation is performed by inserting three groups of instructions. The first group of the instructions is used to declare an array *counter* to record the number of specific locations that have been executed, as described in step 3 of the Execution Analysis algorithm. The second group instructions are used to initialize the array *counter* created by the first group instructions. The third group instructions are inserted at all definition and use locations to update the values stored by the elements of the array *counter*. The definition and use locations are indicated by the DUMs defined in section 3.1.4. The instrumented class generated is then invoked by the *Execution Analysis* component using the desired test cases. The final array *counter* is used to compute the execution probability. For example, assume that the component *C* has five locations of definition and use, and there are four test cases. Suppose that, the number of the test cases that executed all five locations is 3,2,4,2 and 3, respectively. Therefore, the execution probability for each location is computed to be 0.75, 0.5, 1, 0.5 and 0.75.

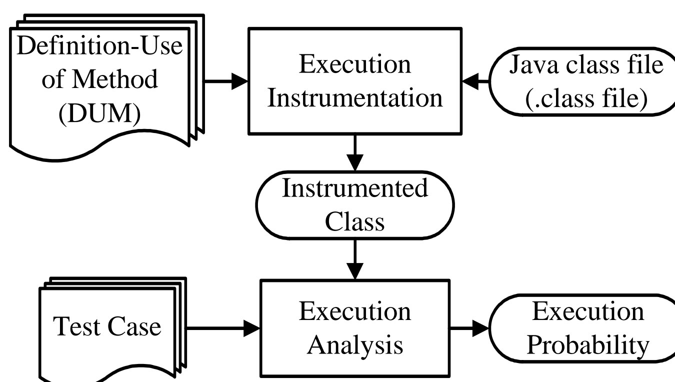

Figure 3.14: Execution Analysis Process

3.3.2 Propagation Analysis

A *data state* is a set of variable values after execution. A definition statement modifies the data state. For example, the data state {(*a, undefined*)} is changed to {(*a, 5*)} after executing the definition statement a=5. The value of variable is used in a use statement. In this thesis, we are interested in faults of data state, namely, *incorrect data state*. The incorrect data state is classified into two types. The first type, *incorrect definition*, occurs when a variable value is defined incorrectly. The second type, *incorrect use*, occurs when a variable value is used incorrectly. Our propagation analysis estimates the probability that an incorrect data state caused by a faulty location will propagate to the output. The incorrect data state is generated by injecting a fault into the data state. This is called a *data state mutation*. This generation process is similar to the mutation process in mutation analysis [22]. Base on incorrect data state, we define two operators of data state mutation: *Definition Replacement* and *Use Replacement*. Definition Replacement operator creates a mutant by redefining the variable value. Use Replacement operator creates a mutant by changing the used variable value to an incorrect value. The data state mutation creates mutants by applying data state mutation operators at definition and use locations at bytecode level. At a definition location, the Definition Replacement operator is applied. At a use location, the Use Replacement operator is applied. For example, at the definition location *l*, the variable *x* is mutated to generate the mutant by redefining the value of variable *x*. Each mutant is executed and the execution output is checked against the original output. If the execution outputs from the original are different those from the mutant, this mutant is considered “killed” by the test. In this thesis, we assume that every mutant could be killed by some tests, therefore, equivalent mutants would not be considered. The propagation probability of a location *l*, *P*(*l*), is the percentage of executions that the execution output of mutant of *l* differs from that of the original program. The algorithm for finding a propagation probability of a particular location is as follows:

1. Create a mutant by data state mutation for a definition or use location *l.*
2. Initialize an integer variable *kill* to 0.
3. Execute the mutant by a test case.

4. Compare the execution output with the output of the original class on the same test case. If the outputs differ, propagation has occurred and *kill* is incremented.
5. Repeat algorithm steps 3 to 4 *p* times, each time with a different test case.
6. Divide *kill* by *p* to calculate the propagation probability of the mutant of location *l.*

The process of propagation analysis was implemented as shown in Figure 3.15. The process consists of two major components: the *Mutation Generator* and the *Propagation Analysis*. The *Mutation Generator* modifies the original java .class file by instrument certain instructions into it and produces mutants by using the data state mutation technique described previously. A mutant is generated for each location of variable definition and use. Our implementation tool generating mutants supports five data types of java, *int*, *double*, *string*, *Boolean* and array of type *int*. Table 3.5 presents the values used to mutate each data type. For example, to produce a mutant at a use location of *int* variable *total*, instructions are inserted at the use location. The inserted instructions change the used *total* value to *MinInt*. The *Propagation Analysis* component then invokes each mutant against the desired test cases. The detail of invoking a mutant against a test case of this component is shown in Figure 3.16. In this figure, the execution output of the mutant against the test case is compared with the execution output of the original class against the same test case. The result of this comparison is used to generate the propagation probability. For example, assume that the component *C* has five locations of definition and use, and four test cases. Applying the data state mutation technique, five mutants were generated for this component. Each mutant was executed four times, each time against each test case. Assume further that, the numbers of different execution outputs for each respective definition and use location are 2, 1, 2, 3, and 3, for mutant execution. Therefore, the propagation probabilities of all locations are 0.5, 0.25, 0.5, 0.75 and 0.75, respectively.

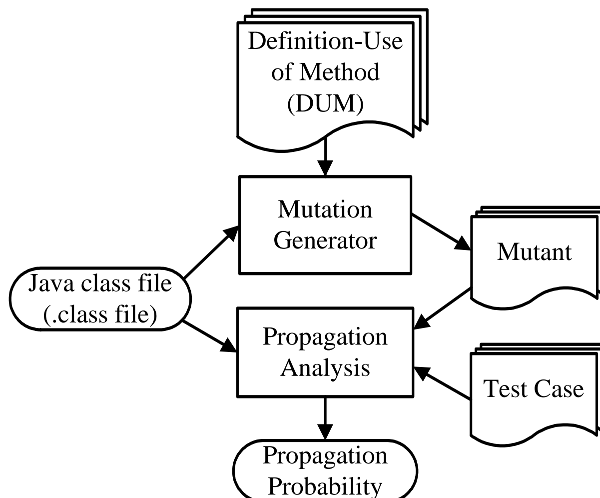

Figure 3.15: Propagation Analysis Process

Table 3.5: Values to Mutate Each Data Type

Data Type	Values to mutate
Int	MinInt
Double	MinDouble
String	Null
Boolean	Inversion of state (true -> false, false -> true)
Array of type Int	MinInt

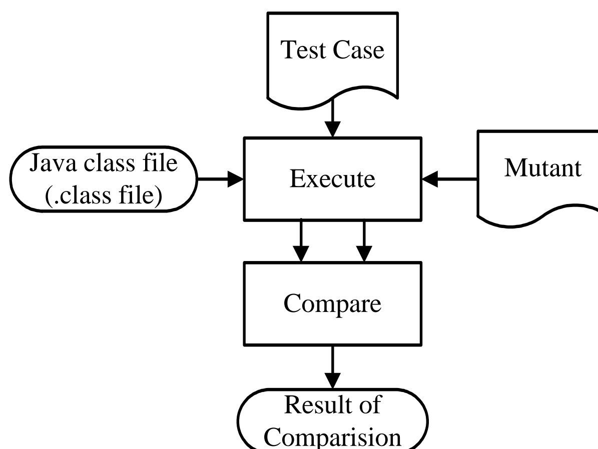

Figure 3.16: The Detail of Execution in Propagation Analysis Process

To compute the testability for each location *l*, the testability of location *l*, *T*(*l*), is the execution probability, *E(l)*, multiplied by the propagation probability, *P(l)*. The testability measure of a class-component (*T*) is the average testability of all locations. If a component has *k* locations, the testability measure is calculated as follows:

$$T = \frac{\sum_{i=1}^{k} T(i)}{k} \quad , \; 1 \le i \le k$$

CHAPTER IV
CASE STUDY

The goal of this thesis is to increase and measure the class-component testability by using the component information analyzed at the bytecode level. This chapter demonstrates our proposed methods. Section 4.1 presents an effectiveness of our increasing class-component testability method, how fault can be easily revealed and failures can be easily detected by using our method. Section 4.2 shows the effectiveness of our testability measurement. By the testability measurement, a class-component with high fault revealing ability should be indicated the high testability measure.

4.1 Increasing Testability Experiment

This section shows the effectiveness of our increasing class-component testability process. As discussed in Section 3.2, increasing controllability and observability present increasing class-component testability. To increase controllability, we support ways to generate test cases based on *All-coupling-uses*. To increase observability, the observability probes are added in order to trace and assert internal state variables. The increased class-component testability makes faults easier revealing.

The mutation testing is used to evaluate the fault detection ability that results from our increasing class-component testability process. Mutation is widely considered to be one of the strongest testing techniques, and is often used as a "gold standard" against which to evaluate other testing techniques. It has been used as a way to induce faults into the program for empirical fault studies in dozens of testing papers [27, 28, 29]. Andrews et al. [27] recently studied the direct question of whether mutation-like faults are valid in studies like this, and found that they are. This supports older evidence [30], which found that tests that detect mutation-like faults are good at detecting more complicated faults.

As explained about mutation analysis in background section, Mutation analysis works by modifying copies of the original program or component. *Mutants* are created by making copies of the original version, then inducing each mutant change into a unique copy. Tests are run on the original version, and then each mutant version. If the results from an original version are different from the results from a mutant version, the mutant is said to be killed by the tests.

Figure 4.1 shows a family of data flow testing criteria proposed by Rapps and Weyuker [16]. The criteria relationships identify that *All-du-paths* is a stronger criterion than *All-uses*. *All-uses* criterion requires a test to cover a path from each definition to reachable uses, whereas *All-du-paths* criterion requires the coverage of all possible definition-use paths. Note that, the definition-use pairs of *All-uses* criterion is the subset of definition-use pairs of *All-du-paths* criterion.

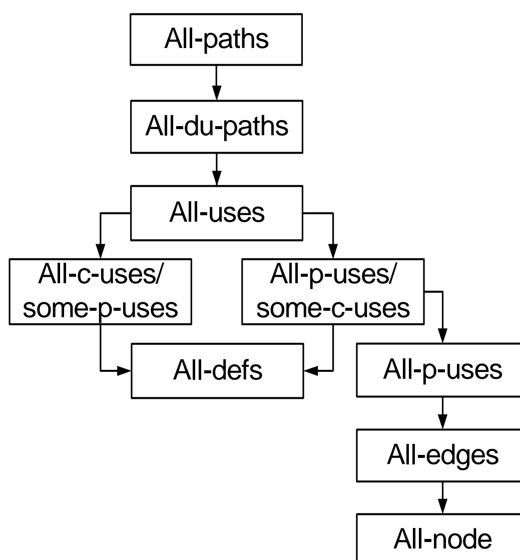

Figure 4.1: A Family of Data Flow Testing Criteria [16]

This thesis generates test cases based on *All-coupling-uses* criterion as explained in subsection 3.2.1. This criteria requires, for each coupling-def, at least one test case executes a path from the definition to each reachable coupling-use. Both *All-coupling-uses* and *All-uses* criteria consider on the paths from each definition to reachable uses. The specific of *All-coupling-uses* criterion is it only considers on the definition-use paths of a variable between the last definition in a method and the first

use in other methods. Consequently, the definition-use pairs of *All-coupling-uses* criterion are the subset of definition-use pairs of *All-uses* criterion. As stated before, the definition-use pairs of *All-uses* criteria are the subset of definition-use pairs of *All-du-paths* criterion. Therefore, the definition-use pairs of *All-coupling-uses* criterion are the subset of definition-use pairs of *All-du-paths* criterion. In other words, the *All-du-paths* requires more complete paths than *All-coupling-uses*. To show effectiveness of our test cases, this experiment uses the test cases generated according to the *All-du-paths* criterion to compare the fault detection ability with our test cases. The detail of experiment is described in the following subsection.

4.1.1 Experimental Setting and Results

The experiment proceeded in six steps:

(1) Prepare classes to test,

(2) Generate a set of test cases following our approach, *All-coupling-uses*, for each class,

(3) Generate a set of test cases to satisfy *All-du-paths* coverage for each class,

(4) Generate the mutants for each class,

(5) Run each set of test cases on the original and each mutant version, and

(6) Compute the fault detection ability of each set of test cases.

The details of each step are explained as following.

We used five subjects, the vending machine class (VendingMachine) and four classes from a data structure package, StackArr, QueueArr, BinaryHeap and ArrList. Table 4.1 shows the description of each class. Column “SLOC” shows the source lines of code. Column “#Instructions” shows the number of bytecode

instructions. Column “#Variables” shows the number of variables. Column “#Methods” shows the number of methods. Following our proposed method as explained in subsection 3.2.1, a set of test cases was generated to satisfy our All-coupling-uses criterion for each class. These are called *All-coupling-uses* tests (*ACU* tests). The ACU tests were duplicated, and observations of internal states were added by observability probes as explained in subsection 3.2.2. These are called *All-coupling-uses with observability* tests (ACU-O tests). The test generation process for ACU and ACU-O tests is shown in Figure 4.2. A dashed arrow (“--->”) is used to represent omitted methods. Also, a set of test cases for each class were generated to satisfy the *All-du-paths* criterion (*AllDU* tests). Table 4.2 shows the number of test cases of ACU and AllDU tests for each class. The obvious observation is that in all classes, AllDU tests require more test cases than ACU tests. This is consistent with the criteria relationship [16] that if a more stringent criterion is chosen, the number of definition-use pairs that must be tested increases. The number of test cases of ACU-O tests is equal to ACU tests, but ACU-O tests were with observability probes. The observability probes help to observe internal state variables during testing.

Mutants were generated for each class by making faults at the definition and use locations, called **data state mutation**. The data state mutation process was performed at bytecode level same as the *Mutation Generator* module described in subsection 3.3.2. We assume that the five subject classes are fault-free. The discovered faults are only from mutants. The number of mutants generated for each class is shown in Table 4.3. Each original class and corresponding mutants were executed against ACU, ACU-O and AllDU tests. A mutant is said to be killed by a set of test cases, if the execution results of the mutant are different from the execution results of the corresponding original class. For example, if the results executed against ACU tests of the VendingMachine class and a mutant are different, the mutant is killed by ACU tests. The killed mutant means a fault is detected. The fault detection scores were computed in terms of the number of faults detected for each class.

Table 4.1: The Description of Each Class Used in the Experiment

Class Name	SLOC	#Instructions	#Variables	#Methods
VendingMachine	40	118	4	5
StackArr	58	140	3	9
QueueArr	59	151	5	8
BinaryHeap	61	201	3	8
ArrList	58	229	4	9

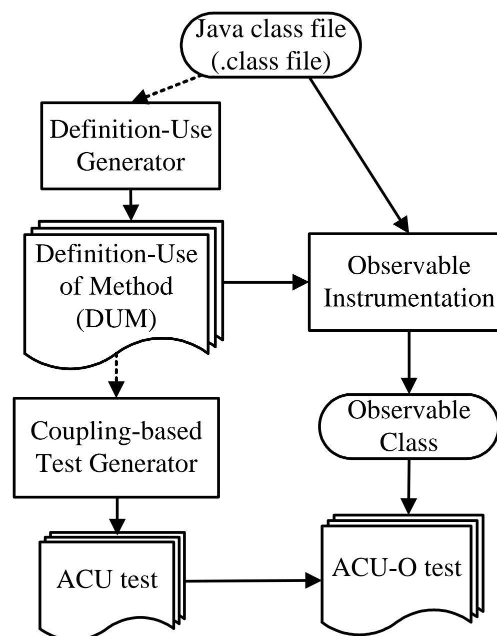

Figure 4.2: The Test Generation Process for ACU and ACU-O

Table 4.2: The Number of Test Cases of ACU and AllDU tests

Class Name	ACU	AllDU
VendingMachine	13	22
StackArr	25	25
QueueArr	22	27
BinaryHeap	22	39
ArrList	58	68

Table 4.3: The Number of Mutants for Each Class

Class Name	#Mutants
VendingMachine	18
StackArr	13
QueueArr	21
BinaryHeap	25
ArrList	36

Table 4.4 and Table 4.5 show the fault detection ability of ACU, ACU-O and AllDU tests. “#Mutants” indicates the number of mutants for each class. “#Tests” indicates the number of test cases for each class. “Faults Detected” indicates the number of killed mutants for each class. “Detection Increase” indicates the number of increment of detected faults for each class. Table 4.4 shows fault detection ability of ACU and ACU-O tests. The execution results of ACU-O tests found more faults than ACU tests of any classes. The fault detection was increased of any classes. The average increment of fault detection is 73%.

Table 4.5 shows the execution results of AllDU and ACU-O tests. The results provided by this experiment indicate that the fault detection ability of ACU-O tests was increased from the fault detection ability of AllDU tests of any classes. The fault detection ability of ACU-O tests was increased 73% from that of the AllDU tests. We found that the increased fault detection ability of ACU-O tests was mostly affected by proposed observability probes. The observability probes help observe the validity of the internal state variables during testing. For example, consider the VendingMachine class’s mutant that was mutated by changing the used *total* value at line 31 in Figure 3.4, this fault was not directly propagated to the output and this mutant was not killed by AllDU tests. Equipped with the observability probes in ACU-O tests, the incorrect *total* value could be detected. As shown in Table 4.5, the proposed ACU-O tests detect all the faults for four out of five classes. The fault detection ability of ArrList class was not 100% because of the *dd* data flow anomaly that occured when a definition is followed by another identical definition without any intermediate use. For example, from ArrList class, the partial instructions shown in Figure 4.3 created the *dd* data flow

anomaly for the variable *end*. As part of the future work, data flow anomaly detection should be eliminated before testing.

In our experimentation, we can observe that applying our All-coupling-uses criterion (ACU tests) to generate test cases provided the greater number of def-clear path executions satisfying definition-use pairs than applying All-du-paths criterion (AllDU tests). As shown in Table 4.6, the def-clear path executions of ACU tests reached 92% of definition-use pairs, whereas the def-clear path executions of AllDU tests reached 74%.

As a conclusion, our increasing class-component testability process provides an improvement in detecting faults. The process supplies the effective test cases for exercising a class-component.

Table 4.4: Case Study Results on All-coupling-uses and All-coupling-uses with Observability

Class Name	#Tests	#Mutants	ACU		ACU-O		Detection Increase	%Detection Increase
			Faults Detected	% Faults Detected	Faults Detected	% Faults Detected		
VendingMachine	13	18	13	72	18	100	5	39
StackArr	25	13	8	62	13	100	5	63
QueueArr	22	21	15	71	21	100	6	40
BinaryHeap	22	25	6	24	25	100	19	317
ArrList	58	36	21	58	32	89	11	25
Sum	140	113	63		109		46	

Table 4.5: Case Study Results on All-du-paths and All-coupling-uses with Observability

Class Name	#Mutants	AllDU			ACU-O			Detection Increase	%Detection Increase
		#Tests	Faults Detected	%Faults Detected	#Tests	Faults Detected	%Faults Detected		
VendingMachine	18	22	13	72	13	18	100	5	39
StackArr	13	25	8	62	25	13	100	5	63
QueueArr	21	27	15	71	22	21	100	6	40
BinaryHeap	25	39	6	24	22	25	100	19	317
ArrList	36	68	21	58	58	32	89	11	25
Sum	113	181	63		140	109		46	

```
…
31:  aload_0
32:  aload_0
33:  dup
34:  getfield       ArrList.end I (4)
37:  iconst_1
38:  iadd
39:  dup_x1
40:  putfield       ArrList.end I (4)        //definition end
43:  aload_0
44:  getfield       ArrList.array [I (2)
47:  arraylength
48:  irem
49:  dup_x1
50:  putfield       ArrList.end I (4)        //definition end
…
```

Figure 4.3: The *dd* Data Flow Anomaly of Variable *end* of ArrList Calss

Table 4.6: The Number of Definition-use Pairs and The Number of Def-clear Path Executions

Class Name	AllDU		ACU	
	#Definition-use Pairs	**#Def-clear Path Executions**	**#Definition-use Pairs**	**#Def-clear Path Executions**
VendingMachine	27	22	14	13
StackArr	31	25	31	25
QueueArr	29	27	23	22
BinaryHeap	73	39	24	22
ArrList	86	68	61	58
Sum	246	181	153	140

4.2 Measuring Testability Experiment

This section shows the usefulness of our measurement. As mentioned in Section 3.3, testability indicates the difficulty of revealing fault. The high testability shows the high fault revealing ability of components. A fault can be revealed when a program segment that contains the fault is executed and the fault affects the output. Our testability measurement is defined as the product of *execution probability* and *propagation probability*. The execution probability of a location l, $E(l)$, is estimated to be the percentage of test cases that execute that location. The propagation probability

of a location *l*, *P*(*l*), is the percentage of executions that the execution output of mutant of *l* differs from that of the original class. The testability of a location *l*, *T*(*l*), is the execution probability multiplied by the propagation probability. The testability measure of a class-component (*T*) is the average of the testability of all locations.

The goal of this experiment was to determine whether the proposed measurement can obtain the right results on **the tests with high fault revealing ability have the high testability measure**. As shown in Section 4.1, the *All-coupling-uses with observability* tests (ACU-O tests) generated after increasing class-component testability gave higher fault revealing than the *All-coupling-uses* tests (ACU tests) and *All-du-paths* tests (AllDU tests). These three sets of tests are used to evaluate our testability measurement.

4.2.1 Experimental Setting and Results

The experiment proceeded in four steps:

(1) Prepare classes to test,

(2) Generate a set of test cases following our approach, All-coupling-uses, for each class,

(3) Generate a set of test cases to satisfy *All-du-paths* coverage for each class, and

(4) Measure testability of each class with each set of test cases.

The details of each step are explained as following.

As the increasing testability experiment in Section 4.1, we used the five classes, VendingMachine, StackArr, QueueArr, BinaryHeap and ArrList, and the three sets of tests, ACU, ACU-O and AllDU. The numbers of test cases of ACU and AllDU tests is shown in Table 4.2. The number of test cases of ACU-O tests is same as ACU tests but the ACU-O tests is added by observations of internal states. As explained in

Section 3.3, the class-component testability is measured by considering the locations of definition and use. The considered locations of each class are shown in Table 4.7. The “#Locations” column gives the number of definition and use locations for each class.

Table 4.7: The Considered Locations of Each Class

Class Name	#Locations
VendingMachine	18
StackArr	13
QueueArr	21
BinaryHeap	25
ArrList	36

To measure the testability of a class, the class with a set of test cases is performed by the processes of execution analysis and propagation analysis explained in subsection 3.3.1 and 3.3.2, respectively. For example, the testability of vending machine class (VendingMachine) is measured by the AllDU tests. There are 22 test cases for AllDU tests. The considered locations of vending machine class are 18 locations. An execution probability of each location is analyzed by the execution analysis process explained in subsection 3.3.1. Table 4.8 shows the number of test cases executing each location. An execution probability of each location, E(*l*), was calculated. For example, the execution probability of location 18 was 0.68 (15/22).

A propagation probability of each location is analyzed by the propagation analysis process explained in subsection 3.3.2. Appling the data state mutation technique, 18 mutants were generated for vending machine class. Each mutant was mutated at each location. Each mutant was executed 22 times, each execution against each test case of AllDU tests. The number of executions, that the execution output of mutant differs from the original output, was counted as shown in Table 4.9. A propagation probability of each location, P(*l*), was calculated. For example, the propagation probability of location 18 was 0.55 (12/22). For each

location, the execution probability was multiplied by the propagation probability to be the testability of the location, T(*l*). For example, the testability measure of location 18 was 0.37. The average of the testability of all locations was computed to be the testability of vending machine class.

Table 4.8: The Number of Test Cases Executing Each Location of Vending Machine

Location	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
#Test Cases	22	22	22	22	16	16	3	17	17	13	12	8	8	8	8	17	15	15

Table 4.9: The Number of Executions that the execution output of mutant differs from the original output of Vending Machine

Mutated Location	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
#Executions	0	15	0	12	13	13	2	12	4	0	8	0	0	8	8	17	12	12

Similarly, the testability of vending machine class was also measured by using the ACU and ACU-O tests. Table 4.10 shows the execution probability, E(*l*), propagation probability, P(*l*), and testability measure, T(*l*), for each location of AllDU, ACU and ACU-O tests.

Table 4.11 shows the testability measure of each class by using AllDU, ACU and ACU-O tests. The “#Mutants” column gives the number of mutants created by data state mutation technique for each class. The “#Tests” is the number of test cases for each class. The “Testability” is the testability measure for each class. The testability measure results of ACU-O tests were higher testability measure than ACU and AllDU tests of any classes. These results support the results from the section 4.1 that the ACU-O tests gave higher fault revealing than the ACU and AllDU tests. The experiment shows that our testability measurement can be used to indicate the fault revealing ability of component.

Table 4.10: The Testability Measure of Vending Machine Class

Location	AllDU			ACU			ACU-O		
	E(*l*)	P(*l*)	T(*l*)	E(*l*)	P(*l*)	T(*l*)	E(*l*)	P(*l*)	T(*l*)
1	1.00	0.00	0.00	1.00	0.00	0.00	1.00	0.31	0.31
2	1.00	0.68	0.68	1.00	0.54	0.54	1.00	0.69	0.69
3	1.00	0.00	0.00	1.00	0.00	0.00	1.00	0.15	0.15
4	1.00	0.55	0.55	1.00	0.38	0.38	1.00	0.62	0.62
5	0.73	0.59	0.43	0.69	0.46	0.32	0.69	0.69	0.48
6	0.73	0.59	0.43	0.69	0.46	0.32	0.69	0.54	0.37
7	0.14	0.09	0.01	0.15	0.08	0.01	0.15	0.15	0.02
8	0.77	0.55	0.42	0.62	0.38	0.24	0.62	0.46	0.28
9	0.77	0.18	0.14	0.62	0.23	0.14	0.62	0.62	0.38
10	0.59	0.00	0.00	0.46	0.00	0.00	0.46	0.46	0.21
11	0.55	0.36	0.20	0.38	0.31	0.12	0.38	0.38	0.15
12	0.36	0.00	0.00	0.31	0.00	0.00	0.31	0.31	0.09
13	0.36	0.00	0.00	0.31	0.00	0.00	0.31	0.08	0.02
14	0.36	0.36	0.13	0.31	0.31	0.09	0.31	0.31	0.09
15	0.36	0.36	0.13	0.31	0.31	0.09	0.31	0.31	0.09
16	0.77	0.77	0.60	0.62	0.62	0.38	0.62	0.62	0.38
17	0.68	0.55	0.37	0.62	0.38	0.24	0.62	0.62	0.38
18	0.68	0.55	0.37	0.62	0.38	0.24	0.62	0.62	0.38
Average Testability			0.25			0.17			0.28

Table 4.11: The Testability Measure of Each Class

Class Name	#Mutants	AllDU		ACU		ACU-O	
		#Tests	Testability	#Tests	Testability	#Tests	Testability
VendingMachine	18	22	0.25	13	0.17	13	0.28
StackArr	13	25	0.18	25	0.18	25	0.37
QueueArr	21	27	0.21	22	0.19	22	0.43
BinaryHeap	25	39	0.10	22	0.07	22	0.18
ArrList	36	68	0.09	58	0.07	58	0.17

CHAPTER V
SUMMARY AND FUTURE WORKS

In this chapter, we conclude our thesis and present some directions for the future work.

5.1 Summary

This thesis presented a new analysis technique to collect control and data information of a class-component at the bytecode level. The collected information is used to increase and measure class-component testability.

The contributions of this thesis are listed below:

- We provided an analysis method and corresponding tool to automatically gather control flow and data flow information. This information is represented by an intermediate graph, CDFG, for each method in a class-component. The product of the analysis method is CDFGs of a class-component.
- Instead of the source code analysis mandated by the conventional approach, our analysis method processed at bytecode instructions. At bytecode level, we consider a dimensional variable as a unique storage location. A dimensional array is viewed as one variable and does not differentiate values between individual elements.
- We implemented preliminary tools to support:
  - Instance fields
  - Fields of primitive type, int, double, boolean and string
  - An array of int
- We used an open source tool from Apache/Jakarta [12] called BCEL (Byte Code Engineering Library) to analyze and instrument bytecode instructions.

- We proposed a method to increase testability of a class-component. To increase testability, CDFGs are used with All-coupling-uses criteria to supply test cases for testing of class-components (increasing controllability), and to make it easier to observe internal state variables during testing (increasing observability).

- We provided a process and corresponding tool for generating test cases. The process generates test requirements for a class-component. Then, the test requirements are used to generate test cases for exercise the component as necessary. The corresponding tool was implemented to automatically generate test requirements. The test cases were manually initialled and modified, and automatically evaluated for satisfying test requirements.

- We provided a process, called *observability probes*, and respective tool for observing internal state variables during testing. The respective tool was implemented to automatically insert tracking mechanisms that recode and assert internal state variables at runtime.

- The CDFGs represented information of a class-component are also used to predict the ease (or difficulty) of component testing, called *testability measurement*. The testability measurement concentrates on the fault revealing ability of a class-component. The class-component testability is measured by analyzing execution probability and propagation probability. The execution probability is the percentage of faulty locations executed. The propagation probability is the percentage of faulty locations for which a test case caused incorrect output.

- We implemented the corresponding tool for supporting the testability measure. This tool automatically instruments a class-component to recode and compute the percentage of executed locations at runtime. The tool also automatically mutates a class-component to product faulty versions, called mutants. The

mutants are executed and the execution outputs are used to compute the propagation probability.

5.2 Future Works

The analysis approach described in this thesis only supported instance fields. Future work should include a process supporting the analysis of class fields. The analysis tool should be extended to perform the complete analysis for individual elements and other types of array.

A few parts of tools described in this thesis are only partially automated. Future efforts will focus on improving the tools to automate all processes.

This thesis focuses on intra-class method calls, but extending the inter-class method calls are straight forward. However, the consideration of multiple classes is more complex when class hierarchies with dynamic type binding and polymorphism are used. This is an issue for future work.

Component-based (CB) software development is currently in widespread use. A CB system is built by assembling already existing components, which need to be retested in the new environment. Another important future work should investigate whether our approach is readily applicable to all component-based (CB) software, not just java.

REFERENCES

[1] A. V. Aho, R. Sethi and J. D. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley Publishing Company, Reading, MA, 1986.

[2] B.-Y. Tsai, S. Stobart, and N. Parrington. Employing data flow testing on object-oriented classes. The IEE Proc. – Softw. 148, 2 (April 2001): 56-64.

[3] M. J. Harrold and M. L. Souffa. An incremental approach to unit testing during maintenance. Proceedings of IEEE/ACM Conference on Software Maintenance, Phoenix, Arizona, USA, 1988: 362-367.

[4] T. Alexander and A. J. Offutt. Analysis Techniques for Testing Polymorphic Relationships. Proceedings of the Thirtieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS30 '99), IEEE Computer Society, Santa Barbara CA, 1999: 104-114.

[5] B. Beizer. Software Testing Techniques. Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.

[6] A. Abdurazik, and J. Offutt. Generating test cases from UML Specifications. Technical report ISE-TR-99-105, Department of Information and Software Engineering, George Mason University, Fairfax VA, 1999. http://www.ise.gmu.edu/techrep/.

[7] L. Gallagher, J. Offutt and A. Cincotta. Integration testing of object-oriented components using finite state machines. The Journal of Software Testing, Verification, and Reliability, in press - published online (January 2006).

[8] H. S. Hong, Y. R. Kwon, and S.D. Cha. Testing of Object-Oriented Programs Based on Finite State Machines. The Asia-Pacific Software Engineering Conference (ASPEC95), Brisbane, Australia, December 1995: 234-241.

[9] E. Weyuker. Testing Component-based Software: A. Cautionary Tale. IEEE Software 15, 5 (Sept/Oct 1998): 54-59.

[10] J. M. Voas, and K. W. Miller. Software testability: The new verification. IEEE Software 12, 3 (May 1995): 17-28.

[11] Y. Wang, G. King, M. Fayad, D. Patel, I. Court, G. Staples, and M. Ross. On Built-in Test Reuse in Object-Oriented Framework Design. ACM Journal on Computing Surveys 32, 1 (March 2000): 7-12.

[12] Apache Software Foundation. BCEL: Byte Code Engineering Library. Part of the Apache/Jakarta project, 2002-2003. http://jakarta.apache.org/bcel/.

[13] T. Lindholm and F. Yellin. The Java™ Virtual Machine Specification, Addison Wesley, second edition, 1999.

[14] Y. G. Kim, H. S. Hong, S. M. Cho, D. H. Bae, and S. D. Cha. Test case generation from UML state diagram. IEE Proceedings – Software 146, 4 (August 1999): 187-192.

[15] J. Laski, and B. Korel. A data flow oriented program testing strategy. IEEE Transaction on Software Engineering 9, 3 (May 1983): 347-354.

[16] S. Rapps, and E. J. Weyuker. Selecting software test data using data flow information. IEEE Transaction on Software Engineering 11, 4 (April 1985): 367-375.

[17] Z. Jin, and J. Offutt. Coupling-based criteria for integration testing. The Journal of Software Testing, Verification, and Reliability 8, 3 (September 1998): 133–154.

[18] IEEE Standard Glossary of Software Engineering Technology, ANSI/IEEE 610.12, IEEE Press (1990).

[19] R. V. Binder. Design for testability with object-oriented systems. Communications of the ACM 37, 9 (September 1994): 87-101.

[20] R. Freedman. Testability of software components. IEEE Transactions on Software Engineering 17, 6 (June 1991): 553–563.

[21] J. Gao, H.-S. Tsao, and Y. Wu. Testing and Quality Assurance for Component-based Software. Artech House, Inc, MA, 2003. ISBN 1-58053-480-5.

[22] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach for integration testing. IEEE Transactions on Software Engineering 27, 3 (March 2001): 228–247.

[23] M. J. Harrold, and G. Rothermel. Performing data flow testing on classes. The ACM SIGSOFT Foundation of Software Engineering, New Orleans, LA, December 1994: 154–163.

[24] T. J. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering SE-2, 4 (December 1976): 308-320.

[25] J. M. Voas. PIE: a dynamic failure-based technique. IEEE Transactions on Software Engineering 18, 8 (August 1992): 717-727.

[26] M. J. Harrold, A. Orso, D. Rosenblum, G. Rothermel, M. L. Soffa, and H. Do. Using component metadata to support the regression testing of component-based software. The IEEE International Conference on Software Maintenance, Florence, Italy, November 2001: 154-163.

[27] J.H. Andrews, L.C. Briand, & Y. Labiche. Is mutation an appropriate tool for testing experiments?. The 27th International Conference on Software Engineering, St. Louis Missouri, USA, 2005: 402-411.

[28] R. A. DeMillo, R. J. Lipton and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer 11, 4 (April 1978): 34-41.

[29] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. IEEE Transactions on Software Engineering 17, 9 (September 1999): 900-910.

[30] J. Offutt. Investigations of the software testing coupling effect. ACM Transactions on Software Engineering Methodology 1, 1 (January 1992): 3-18.

APPENDIX

APPENDIX A

PUBLICATIONS

A.1 International Journal

1. Kansomkeat, S., Offutt, J., and Rivepiboon, W. Bytecode-based Analysis for Increasing Class-Component Testability. ECTI-CIT Transactions on Computer and Information Technology, Volume 2, Number 2, November 2006. http://www.ecti.or.th/~editoreec/index_eec.htm
2. Kansomkeat, S., Offutt, J., and Rivepiboon, W. Analysis for Class-Component Testability. WSEAS Transactions on Computers, Issue 2, Volume 5, 2006: 352-358, ISSN 1109-2750.

A.2 International Conference

1. Kansomkeat, S., Offutt, J., and Rivepiboon, W. Class-Component Testability Analysis. The proceeding of the 5th WSEAS International Conference on Software Engineering, Parallel & Distributed Systems, 2006.
2. Kansomkeat, S., Offutt, J., and Rivepiboon, W. Increasing class-component testability. Proceedings of the IASTED International Conference on Software Engineering (SE 2005), 2005: 156-161.
3. Kansomkeat, S., and Rivepiboon, W. Component specification to test component-based software. Proceedings of the International Society for computers and Their Applications, 2004: 282-285.
4. Kansomkeat, S., and Rivepiboon, W. Automated-generating test case using UML statechart diagrams. Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information, 2003: 296-300.

BIOGRAPHY

Name Miss Supaporn Kansomkeat

Sex Female

Date of Birth August 8, 1969

Education:

2006 Ph.D. in Computer Engineering, Chulalongkorn University

1995 M.Sc. in Computer Science, Prince of Songkla University

1991 B.Sc. in Mathematics, Prince of Songkla University